98 FAN BOOKS 2

98マシン語

初めて学ぶマシン語の集中レッスン

藤木 文彦 著

技術評論社

カバーイラスト■一木　英夫
カバーデザイン■エムワン工房

98 FAN BOOKS 2

はじめて学ぶマシン語の集中レッスン

98マシン語

藤木文彦　著

技術評論社

はじめに

本書はNEC PC-9801をとおして，16ビットCPU 8086のマシン語を学習しようとする方のための本格的な入門書です．

PC-9801が世に出てはや2年．世の中は8ビットから16ビットの時代へと移行しつつあります．そうした時代にあって，16ビットCPUに対する人々の関心も高まってきました．しかしながら，今日まで16ビットCPUのための入門書といわれるものは，ほとんどと言っていいほど見あたりません．多くの解説書は，ある程度の専門知識を持った人のために書かれたものであり，初心者にとっては難解なものばかりでした．

そうしたことから，「16ビットCPUは難しい」という誤解が生じるにいたっているのではないでしょうか．しかしそんなことはありません．なぜ今まで16ビットCPUが，多くの人々にとって難しいものであったのか．それは，平易に書かれた入門書がなかったからにほかならないのです．しかも通り一遍の解説をした入門書では，残念ながら8086を理解することはできません．

そこで生まれたのが本書です．本書には今までの入門書にはない数々の特徴があります．以下にそれらの特徴を示しましょう．

- マシン語やハードウェアについての予備知識をいっさい必要としない．
- CP／M-86やMS-DOSは使用せず，本体とDISK-BASICさえあれば，誰にでも学習が始められる．
- ステップ学習方式を取り入れ，読み進んでいくうちに自然とマシン語が理解されるように書かれている．
- 命令の解説については，やさしい導入から始めて，正確な使用法を身につけるまで，順に説明がなされている．
- 新しい概念や考え方には，くどいほど十分に解説をつけた．
- 身近な機械であるPC-9801を操作しながら学習を進めることにより，理解が深められるが，機械がなくても学習ができる．
- 豊富な例題や練習問題によって，ひとつひとつの命令の意味や使用法が確実に理解でき

るようになっている．

- マシン語を修得された方にとっても辞書的に使用できるように，命令の使用法が表になっている．

などがあげられます．さらに，例題や練習問題にはすべてリストを掲げてありますから，PC-9801をお持ちでない方や，他の機種で8086の勉強をしようとする方も学習できるようになっています．

本書では，8086のマニュアルにあるような難解な表現をわかりやすく書き直し，さらに定義や用法についても正確で平易であるように心がけましたので，中，上級の方が文法書として使用されるのにも適していると思います．

また，多くの解説書はCP／M-86等を使用してアセンブリ言語を解説していますが，これが初心者にとってのマシン語学習の障害となっていました．それというのも，CP／M-86上のアセンブラであるASM-86などには，すでにマシン語を知っていることを前提に作られた規則がいくつもあるからです．そうでなくても，マシン語を知るためにはまずCP／M-86を知らなければならない，というのでは本当に大変なことです．

そこで本書では，思い切ってPC-9801本体内蔵のアセンブラを用いました．こうすることにより，初心者にとってマシン語がより一層身近なものとなるのではないでしょうか．もちろん，大きなプログラムを作るためにはASM-86が必要ですが，初心者の作る小さなプログラムには，内蔵アセンブラで十分です．したがって，学習の容易さという点から本書では内蔵アセンブラを用いています．

何事も基礎が肝心です．GDCの使い方は，8087の使い方は，と先走る前に，本書でガッチリ基礎を身につけてください．皆さんが本書を読み通されて，マシン語を自由にあやつり，プログラムを作成できるようになることを願ってやみません．

●本書で学習するにあたって

本書は，マシン語への導入部分とマシン語の解説部分とで構成されています．

第1章，第2章は，マシン語の入門に必要な基礎知識について述べた部分です．マシン語にふれるのはまったく初めてという方は，ここから読み始めてください．ただし第1章は，若干ハードウェアについての解説がありますので，わかりにくい部分があるかも知れませんが，読みとばしてど

んどん先に進んでいって結構です．第2章では，2進数とアセンブリ言語，そしてモニタの使い方について解説していますが，すでに2進数やアセンブリ言語についての知識をお持ちの方も，軽く目を通しておいてください．また，モニタの使い方については熟知しておいたほうが良いと思いますので，よく読んでおいてください．

第3章以降は命令の解説です．本書は読むだけでも理解できるように書かれていますが，できれば機械の前に座って，実際にプログラムを入力してみてください．特に練習問題は，例題を参考にすればできるものばかりですから，できるだけ自分の力でプログラムを作成してみることをお勧めします．なお，プログラムはいきなり機械の前で入力するよりも，紙と鉛筆でしっかりと作成してから入力したほうがよいでしょう．

本書で学習を進めていく際に，もしつまずくようなところがありましたら，そこはとばして次に進んでください．マシン語以外の学習でも言えることですが，直線的に進むだけでは理解できないこともあります．わからないところがあったら前にもどって調べ，それでもだめなら，ひととおり終わってからまた読み直すようにしてください．そうした学習の積み重ねによって，きっと8086のマシン語のすべてを理解することができるようになるでしょう．場合によっては，一度めは通読し，二度めに機械に入力してみる，といった方法も良いかも知れません．

本書は，初心者向けにていねいに解説がなされていますし，問題も豊富ですから，すべてを読み通すには多くの時間を必要とすると思います．しかし途中で投げ出さずに最後まで学習し終えることができれば，必ずマシン語が理解できると信じて，ぜひ最後までやり通してください．

内容については，筆者の力不足ゆえに至らぬ点もあることと思います．読者の方々の御意見，御批判をいただければ幸いと存じます．

なお，本書を執筆するにあたって，多くの助言をいただいた三樹聡氏，本書の発刊に至るまで多大な御尽力をいただいた技術評論者編集部の方々に，この場をかりて厚くお礼申し上げます．

1984年8月

藤木文彦

もくじ

第1章 8086CPUとPC-9801について知ろう

1 PC-9801マイクロコンピュータ ——— 14

1 PC-9801マイクロコンピュータ …… 14
2 PC-9801基本構成 …… 15
3 CPUとメモリの働き …… 18

2 8086の特徴 ——— 23

1 16ビットCPUの生立ち …… 23
2 8086のレジスタ構成 …… 24
3 1Mバイトのメモリとセグメント …… 27
4 セグメントに分ける意義 …… 29
5 CPUの内部構成 …… 30
6 TTLの基礎知識 …… 33
7 8086の端子の働き …… 34

第2章 マシン語の基礎知識

1 2進数と16進数 ——— 42

1 10進数と2進数 …… 42
2 2進数の加減算 …… 44
3 レジスタと2進数 …… 45
4 2進数と16進数 …… 46
5 負の数と補数表現 …… 47

もくじ

2 機械語とアセンブリ言語 50

1 人間と機械の橋渡し・アセンブリ言語 50
2 機械は命令をどのように理解するか 52
3 命令の種類と命令長 53
4 ハンド・アセンブル 54

3 モニタの使い方 55

1 モニタとアセンブラ 55
2 ディスクを用意しよう 56
3 モニタ・コマンドの使い方 57

第3章 転送命令とアドレス指定

1 データ転送命令 76

1 データ転送命令の記述法 76
2 メモリ・レジスタの指定法 77
3 格納のされ方 77
4 データセグメントの設定 78
5 メモリ転送と逆ワード 82
6 バイトとワード 83

2 アドレッシング・モード 87

1 アドレッシング・モードの種類 87
2 アドレッシング・モードを用いるときの注意 90

もくじ

第4章 演算命令を学ぼう

1 加算命令 —— 96

1 ADD命令 …… 96
2 桁上がりとキャリーフラグ …… 98
3 ADC命令 …… 100
4 XCHG命令 …… 103

2 減算命令 —— 107

1 SUB命令 …… 107
2 SBB命令 …… 108

3 符号付演算 —— 111

1 補数表現と加減算 …… 111
2 8ビットデータの16ビットへの変換 …… 113
3 符号付演算と符号拡張命令 …… 115

4 乗除算命令 —— 120

1 乗算命令（MUL・IMUL） …… 120
2 除算命令（DIV・IDIV） …… 122

5 ランダムテクニックとフラグレジスタの働き —— 130

1 ランダムテクニック …… 130
2 フラグレジスタの働き …… 131

もくじ

第5章 繰り返し命令と入出力

1 比較・分岐命令 ——136

1 繰り返しの必要なプログラム……136
2 分岐命令とフラグレジスタ……141
3 分岐命令の種類……144
4 無条件分岐命令……148
5 無条件分岐命令とアドレス指定法……150
6 条件分岐命令のアドレス指定法……153
7 相対分岐型と絶対分岐型……153
8 INC命令・DEC命令……158
9 比較命令(CMP)……158
10 相対アドレスの計算法……163

2 BASICで機械語を ——166

1 メモリへの書き込み・読み出し(PEEK・POKE文)……166
2 機械語ルーチンの呼び出し(USR関数)……168

3 サブルーチンとスタック ——175

1 サブルーチン……175
2 CALL命令とRET命令……178
3 サブルーチンとスタック……183
4 スタック動作とスタック・ポインタ……184
5 スタックへの読み書き(POSH・POP)……187
6 サブルーチンがスタックを使用するのは？……191
7 スタックの用途……192

もくじ

第6章 16ビットならではの命令

1 ループ命令 198

1 繰り返しの必要なプログラム …… 198
2 条件判断をともなうループ命令 …… 200

2 ストリング命令 208

1 LODS，STOS命令 …… 208
2 MOVS命令 …… 212
3 リピートプリフィックス命令 …… 213
4 転送命令群と比較命令群 …… 217

3 割り込みと入出力 225

1 割り込み処理とは？ …… 225
2 割り込み使用法 …… 226
3 割り込み処理と割り込みベクトル …… 227
4 内部割り込み …… 228
5 特殊な内部割り込み …… 229
6 割り込みの禁止 …… 230
7 割り込み命令による入出力 …… 231

第7章 さらに知識を深めるために

1 論理演算命令とシフト命令 238

1 論理演算命令 …… 238

もくじ

2 シフト・ローテート命令 …… 244
3 乗除算とシフト・ローテート命令 …… 246

2 I／O命令 250

1 I／O命令の使用法 …… 250

3 その他の命令 253

1 2進化10進数 …… 253
2 10進補正命令 …… 254
3 特殊な転送命令 …… 256
4 CPU制御命令 …… 259

巻末 APPENDIX

命令一覧表 262

命令一覧表の読み方

命令一覧表

ハンド・アセンブルとハンド・ディスアセンブル 274

ハンド・アセンブルの方法

ハンド・ディスアセンブルの方法

さくいん 279

第1章

8086CPUとPC-9801について知ろう

1　PC-9801マイクロコンピュータ

2　8086の特徴

1 PC-9801マイクロコンピュータ

1 PC-9801マイクロコンピュータ

PC-9801マイクロコンピュータがこの世に出たのが1982年10月．その後1983年10月には，姉妹機のPC-9801E/Fも発表され，ソフトウェアもますます充実し，いまや16ビット・マイクロコンピュータの中心的な存在となっています．

PC-9801の注目すべき点は，マイクロコンピュータとしては初めて，インテル社の16ビットCPU8086を採用し，マイコンの16ビット化の火ぶたを切ったばかりでなく，現在でもユーザーの数を増やし続けていることにあります．この機種の出現により，いままでオフコンやミニコン以上でなければ行なうことのできなかった大量データの高速処理が可能となり，マイコンの応用範囲が一段と広がることになりました．そのため，PC-9801は，パーソナル・ユースから技術計算用，事務処理用と，幅広い用途に使用されています．

16ビットCPU8086

しかし，このPC-9801の能力は，一般的には十分に活かしきられているとはいえません．確かに，BASICでも十分高速処理ができるし，最近では優れたコンパイラも数多く発表されているため，それらのコンパイラを使用することにより，PC-9801の能力をさらに引き出すことは可能です．ただ，どうしても綿密なプログラムを作る必要のある場合には，やはりマシン語によってプログラムを作らなければならなくなります．

そこで，本節では，マシン語によってプログラムを作るために最低限は知っておきたいハードウェアの構成，次節では，8086CPUの構造について解説したいと思います．

なお，ハードウェアの構成といっても，ごく基本的なことばかりなので，何の予備知識もいりません．気楽に読み進めていってください．

2 PC-9801の基本構成

2-1　マイクロコンピュータの基本構成

マイクロコンピュータが1台の計算機として働くためには，数多くのLSIやICから組み立てられたボードや，CRTディスプレイ，キーボードなど，いろいろなものが必要となります．ここで，マイクロコンピュータの最も基本的な構成について考えてみましょう．

マイコンの基本構成

マイクロコンピュータを構成する部分は，大きく分けると次にあげる3つの部分になります．

① 演算および制御を行なう**中央処理装置(CPU)**

② データを記憶する**メモリ（ROM，RAM）**

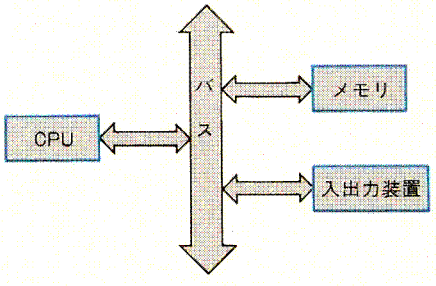
③ 外部とデータをやりとりする**入出力装置（キーボード，CRT，ディスクなど）**

CPUはメモリからプログラムやデータを読み出して処理を続け，必要に応じて入出力装置を通して外部からデータを読み込んだり，外部にデータを送り出したりしています(**図1.1**)．

なお，これらの部分は，互いに**バス**と呼ばれる多数の信号線によって接続され，このバスを通してデータのやりとりが行なわれます．

実際のマイクロコンピュータでは，これらの他に様々な部品が取付けられ，かなり複雑な構成になっていますが，マシン語の入門にはとりあえず先にあげたような基本構成について知っていれば十分です．

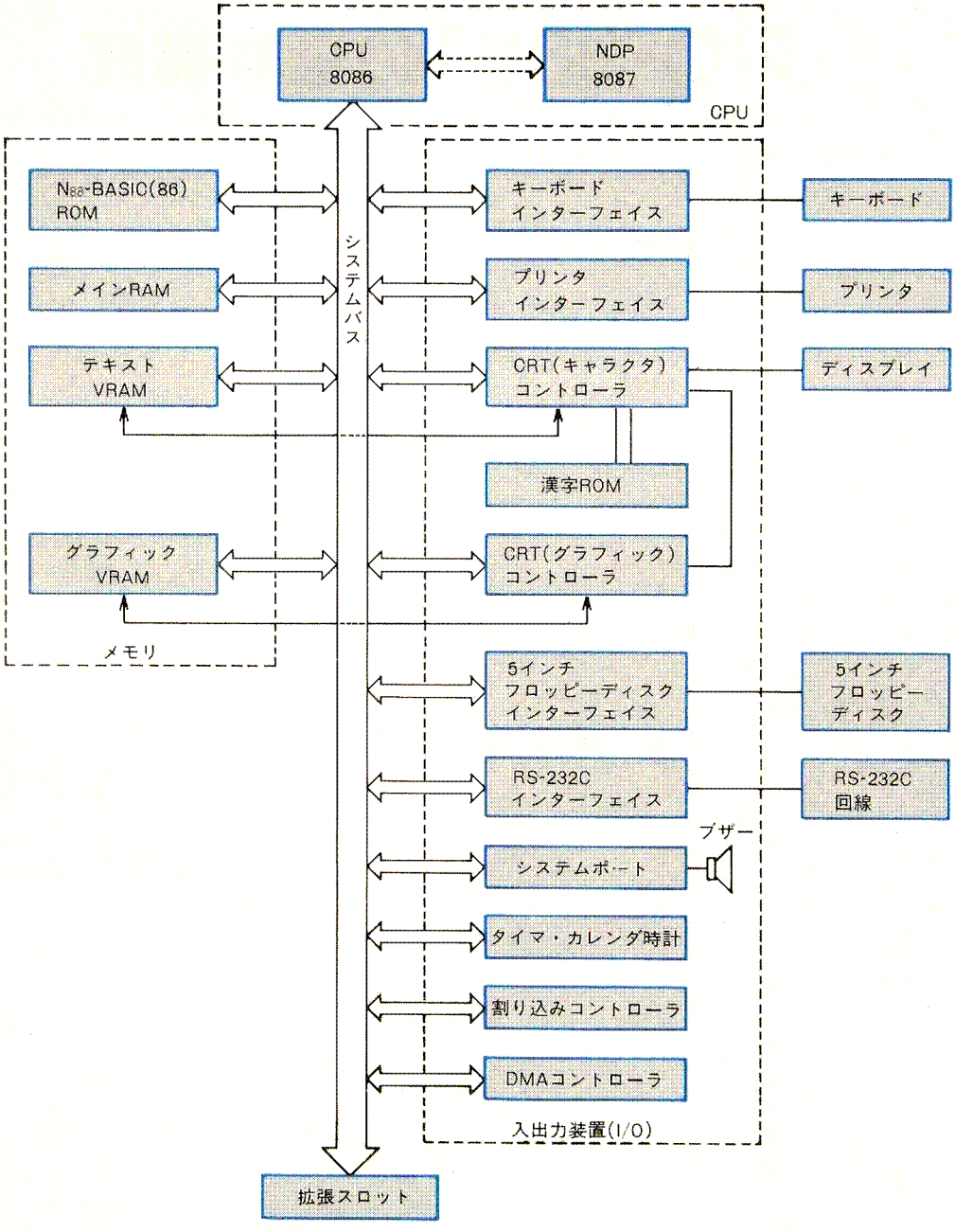
図1.1　マイクロコンピュータの基本構成



2-2　主要装置の機能

次ページ**図1.2**は，PC-9801のブロック図を示したものです．最初のうちは，これらについて覚える必要はありませんが，より進んだプログラムを作るためには，周辺機器をコントロールするためのコマンドなどを知らなければなりません．それらのコマンドについては，後で述べることにしましょう．

図1.2　PC-9801のブロック図



※メモリはアドレスを持ち，データ転送命令などによって直接読み書きできる．

※入出力装置はI/Oポート番号を持ち，IN，OUT命令によって，所定の手続きを踏んでデータを読み書きする．このように入出力装置をI/Oポート番号によって指定する方法をI/Oマップド I/O方式と呼び，入出力装置をメモリ上のアドレスに割り当てる方法を，メモリマップド I/O方式と呼ぶ(68系で使用されている)．

次に各ブロックの機能について簡単にまとめておきました．詳しくは専門書にゆずることにします．また，わからなければ飛ばしていただいてもかまいません．

各ブロックの機能

CPU(中央処理装置)……演算および周辺装置の制御を行ない，コンピュータの中心となる装置．8086という16ビットLSIが使用されている．

NDP(数値データプロセッサ)……関数演算をCPUに代わって高速で行なう装置．8087というLSIが使用できるが，オプションとなっている．

ROM(Read Only Memory)……読み出し専用メモリのことであり，N88BASICおよびモニタなどのシステムプログラムが書かれている．96Kバイトあり，電源を切っても内容は消去されない．

RAM(Random Access Memory)……読み出し，書き込みともにできるメモリのことであり，128Kバイトある．電源を切ると内容は消えてしまう．

キーボード・インターフェイス……キーボードからのデータを変換する装置．

プリンタ・インターフェイス……プリンタに出力するデータを変換する装置．

5インチフロッピーディスク・インターフェイス……5インチフロッピーディスクとデータをやりとりするための装置．

RS232C・インターフェイス……RS-232C規格のデータをやりとりする装置．

システム・ポート……専用の入出力装置を持っていないいくつかの機器と信号をやりとりする（ブザー，DIPスイッチなど）．

割り込みコントローラ……外部機器からの割り込み要求があったとき，優先準位を決め，定められた割り込みを発生させる装置．

DMAコントローラ……CPUを介さずに直接に周辺機器やメモリの間でデータをやりとりする際にタイミングを調整する装置．

CRTコントローラ(C)，(G)……VRAM（後述）上にデータを書き込んだり，VRAM上のデータを変換してCRTディスプレイに表示できる形の信号として送る装置．グラフィック・ディスプレイ・コントローラ(GDC) μPD7220などが使用されている．

テキストVRAM（Video　RAM）……テキスト画面を記憶する専用のRAM.
グラフィックVRAM……グラフィック画面を記憶する専用のRAM.
カレンダ時計……カレンダおよび時計の機能を持ち，必要に応じてデータを送り出す装置.
拡張スロット……その他の周辺装置を取り付けるためのインターフェイスや，拡張RAMなどのボードを入れるスロット.

3 CPUとメモリの働き

3-1　CPUとは何か

先にも述べたとおり，CPUはマイクロコンピュータの中心となる部分であり，「中央処理装置」と呼ばれています．このCPUの機能は，大きく分けて2つあります.

制御機能

ひとつは**制御機能**であり，メモリから命令やデータを読み出したり，メモリにデータを書き込んだり，I/O機器との間でのデータのやりとりを制御したりする機能です．CPUからは，アドレス情報を送り出したり，データのやりとりをしたりするためのデータバスの他に，そうしたデータが正しく目的の周辺機器に伝わるように制御する信号線が何本も出ています．CPUはこれらの信号線を使用して，他のLSIやIC，周辺機器とコミュニケーションを行ないながら，データをやりとりします．なお，命令を実行するために必要な信号の操作はCPUが自動的に行なうので，ユーザーは信号線の制御については考える必要はありません.

演算機能

もうひとつは，命令に従ってデータの演算を行なうという，**演算機能**です．CPUの内部には，レジスタと呼ばれるいくつかの特殊なメモリがあり，メモリ・レジスタ間や，レジスタ・レジスタ間などにおける演算を行ないます．レジスタは，メモリと同様データを記憶するためのものですが，CPUの内部にあり，高速処理が可能です．そのため，演算命令ではレジスタを用いた演算が多く使用されます.

16ビットCPUはバスの幅が16ビット分

以上説明したような機能を，まとめて1個のLSI（中には数個のLSIになっているものもある）にしたものがCPUです．8086は16ビットCPUと呼ばれていますが，それは一度に転送することのできるデータの大きさが16ビット，すなわち，2進数で16桁であるということを意味しています.

なお，8086CPUの詳しい構造と2進数の概念については，次節以下で説明す

ることにします．

3-2　メモリとアドレス

メモリは８ビット単位

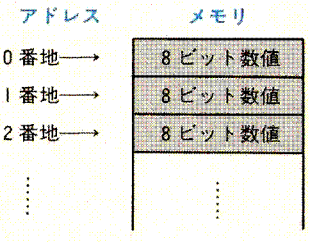
メモリとは，プログラムやデータを記憶するための装置です．プログラムといっても，結局は文字や記号の集まりであり，コンピュータではそうしたものはすべて，ある種の規則に従って数値に変換され，記憶されますから，結局メモリとは数値を記憶するためのものといってよいでしょう．

マイコンでは，数値を記憶するときに，すべて８ビットを単位として行ないます．ビットとは情報の単位であり，８ビットとは，２進数の８桁分のデータ（情報）です（２進数については後で説明します）．そして，指令に従って，データやプログラムをこの８ビットの数値に変換して記憶したり，読み出したりするわけです．しかし，でたらめに記憶していったのでは，後でデータを読み出すときに，どこにあったのかわからなくなってしまいます．そこで，メモリにあらかじめ，順番に番号を付けておきます．この番号のことを**アドレス**，または**番地**と呼びます．こうしておけば，記憶するときにきちんと アドレスを決めておけば，読み出すときに，同じアドレスから読み出せばよいわけです．

アドレス番地

具体的には，メモリに８ビットを単位として，０番地から順にアドレスを付けていきます（**図1.3**）．８ビットのことを１バイトと呼びますから，１バイトにひとつずつアドレスが付くことになります．マイコンは，ほとんどこの１バイトを単位として，データの読み書きを行ないます（なお，本書では「何番地」というときには番地という呼び方を，それ以外の場合にはアドレスという呼び方をしますが，番地とアドレスとはまったく同じ意味です）．

図1.3　メモリとアドレス



1バイト＝8ビット

8086では全部で１メガバイト（＝10^6バイト：１Mバイトと書く(**注１**)）までのメモリが使用できますが，PC-9801ではその約１/3程度を実装しています．実装していない分は，拡張RAMなどを取り付けることにより使用が可能となります．このようにマイコンでは，すべて１バイト（＝８ビット）を基本にしています．8086は16ビットCPUですが，やはり基本は８ビットであり，現在のマイコンはほとんどがそうなっています．ただし８ビットCPUと異なる点は，一度に16ビット，つまり２バイト分のメモリの内容を読み出したり，書き込んだりすることができる点です．この**２バイト**のことを**１ワード**ともいいます．

２バイトを１ワードという

注１） メモリの大まかな大きさを表現するときに，１Kバイト（１キロバイト）とか，１Mバイト（１メガバイト）とかいうが，これは概数であり，実は1024バイトのことを略して１Kバイトと呼んでいる．これは，コンピュータにおいては２の累乗数が基本となっているためであり，$1024=2^{10}$が1000に近い数であることを利用して，このように略す習慣となっている．したがって，たとえば64Kバイト＝64×1024＝65536バイトとなる．さらに，１Mバイト＝1024×1024＝1048576バイトと，数値が大きくなるほど10進数からのずれが大きくなっていくが，実用上問題なく，便利なので，このような表記法が慣習となっている．

したがってCPUは，たとえば0番地の内容と1番地の内容を一度に読み込んで処理することができるわけです．このあたりが8ビットCPUとの違いです．

3-3　PC-8801のメモリ構成

さて，8086では1Mバイトまでのメモリを使用できることがわかりましたが，それではPC-9801では，そのメモリをどのように使用しているのでしょうか．

先に説明したように，メモリといってもシステムプログラムの書かれているROMや，データの読み書きを行なうRAM，ディスプレイ用のVRAMなど，いくつかの種類があります．そこで，これらのROMやRAMをそれぞれ特定のアドレスに配置する必要があります．メモリの用途を表にしたものをメモリ・マップと呼びます．図1.4は，PC-9801のメモリ・マップを示したものです．

メモリの位置関係を表現するとき，0番地に近い方を下位（低位）アドレス，大きい方を上位（高位）アドレスと呼びます．

メモリ・マップを書くとき，0番地を上側にする場合と，下側にする場合とがあります．NECのマニュアルでは0番地が下側になっていますが，一般には0番地を上に持ってくる方が，プログラムの処理の流れと一致していて，わかりやすいので，本書でも0番地の方を上に持ってきます．

RAMは先頭から128Kバイト

図1.4をみるとメモリの先頭部分にRAMがあり，未実装部をはさんでVRAMが，そして最後にBASIC･ROMが配置されていますね．みなさんが，これからマシン語の勉強をしていくのに使用するのは，この一番先頭のRAMの部分であり，16進アドレスで0～1FFFF番地までです（16進数についての詳しい説明は後でしますが，今後みなさんは一応16進数の表記の仕方程度は知っているものとして，アドレスは16進数を用いて表現します）．

ところで，このRAMの部分は，どこでも勝手に使ってよいのかというと，そうはいきません．RAMもいくつかの部分に分かれていて，それぞれに用途が決まっているので，自由に使える部分は限られています．

メモリマップ

図1.5に，RAM部分の詳しいメモリ・マップを示します．

いろいろな部分がありますね．ざっとみていきましょう．

システムエリアには，割り込みベクトルと呼ばれ8086が割り込み処理（後述）を行なうときに参照するデータと，PC-9801本体および周辺機器がどのような状態であるかを示すデータが書かれます．

RAMは場所によって用途が定まっている

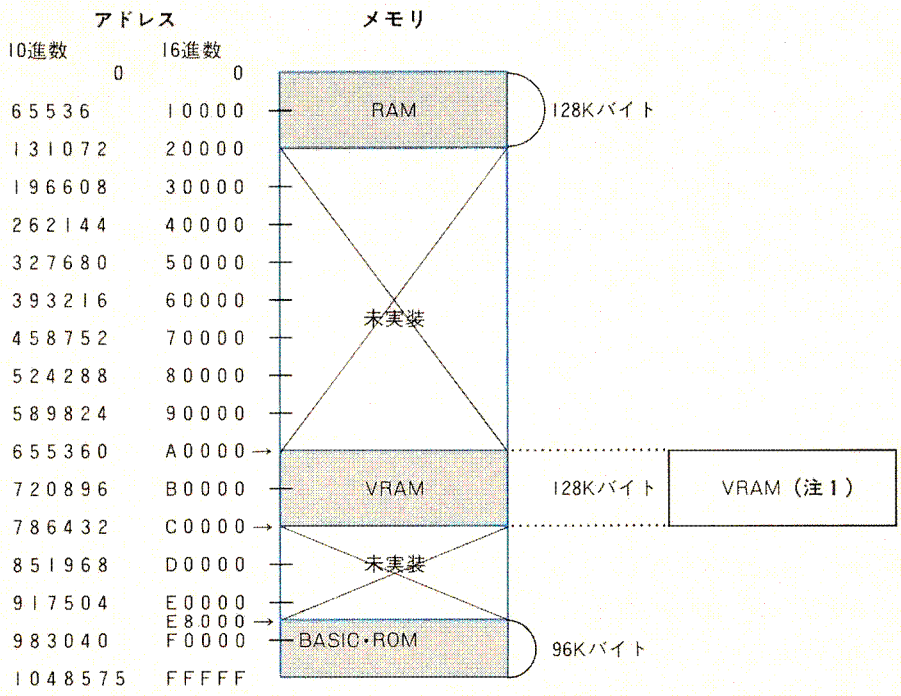
BASICワークエリアとI/Oバッファは，BASICプログラムを実行するときにプログラムを解釈・実行するための作業をする場所，および，周辺機器とやりとりするデータを一時的に蓄えておく場所です．

プログラムテキストエリアは，その名のとおり，BASICのプログラムを格納する場所です．

DISK・CODEエリアは，N88DISK・BASICを使用するために，DISKから読み出されたプログラムで，BASIC・ROMに含まれていないDISK関係の制

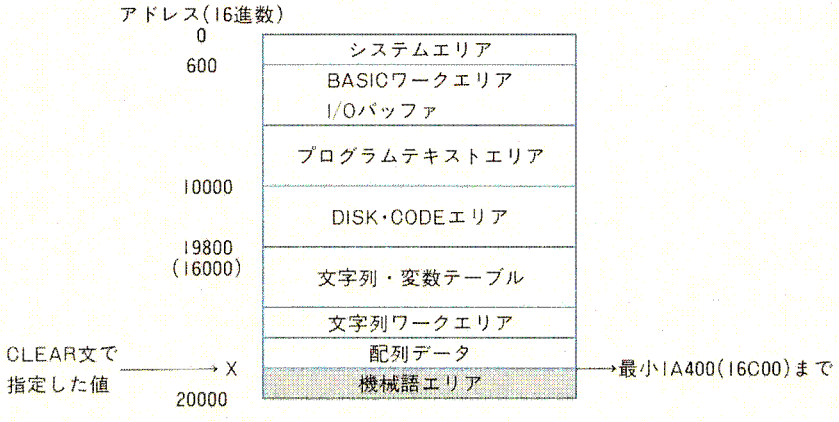
➡22ページに続く

図1.4　PC-9801のメモリ・マップ



注1)PC-9801E/Fのみ．切り換えによって使用できる．通常，ひとつのアドレスにはひとつのメモリしか対応できないが，この部分は特殊なスイッチを切り換えることにより必要な方を選ぶことができるので，このように同一アドレスに複数のメモリを対応させることができる．一度に使用されるのは，選択されている方だけである．

図1.5　日本語DISK-BASICの場合のRAMの使われ方



＊アドレスの書かれていない境界は，必要に応じて変化することを意味する．
＊(　)内は日本語を使用しない場合で，システムディスクおよびメモリスイッチ6で選択する．

御や，モニタ機能の拡張を行なうプログラムです．日本語版の場合には，さらにここに日本語処理を行なうためのプログラムが入るため，通常のN88DISK・BASICよりも大きな領域を必要とします．

文字列・変数テーブルの部分には，文字列や変数のデータが格納されます．

文字列ワークエリアは，文字列データの処理を行なうための作業領域です．

配列データエリアは，配列データを格納する場所です．

さて，最後に機械語エリアですが，この部分が，みなさんがこれからマシン語プログラムを作成していく場所です．このエリアは，使用に先立ちBASICのCLEAR文によって使用する領域を設定しておかなければなりません．通常，機械の電源を入れたり，リセットした直後には，（X）の値は20000（16進数）に設定されています．つまり，機械語エリアはまったく存在しないことになります．そこで，CLEAR文によって，機械語エリアを設定します．しかし，その範囲には制限があり，いくらでも大きくできるわけではありません．具体的には，その直前の配列データエリアなどに，最低限必要なメモリを確保した残りの部分を機械語エリアとして設定できます．その範囲は，

機械語エリアはCLEAR文で確保

N88DISK・BASICを使用したとき……16C00番地以降

N88日本語DISK・BASICを使用したとき……1A400番地以降

となっています．

本書では，N88日本語DISK・BASICを使用した場合を考え，若干の余裕をみて，1B000番地以降を機械語エリアとして使用することにします．

そこで，1B000番地以降を機械語エリアとして使用するためには，

CLEAR, &H1B00↵

と入力します．このとき指定する値は，実際に使用するアドレス（16進5桁で表わされる）のうち，上の4桁だけです．一番下の桁は0とみなされることになっているので，これによって1B000番地以降を機械語エリアとして使用することができるようになります．

このように機械語エリアを設定するということは，BASICが作業領域として使用しない領域を確保するという意味です．逆にいえば，このようにして機械語エリアを設定しないと，BASICが勝手に配列などを格納する場所として使用してしまうということです．機械語エリアを設定しなくても機械語プログラムをRAM上に書くことはできますが，その内容はBASICによって書き換えられてしまう可能性があるわけです．したがって，これから機械語を実際にマイコンに入力して練習する前には，必ずCLEAR文を使用して機械語エリアを確保するようにしてください．

なお，一度CLEAR文によって設定された値は，再設定し直すか，リセットするまで有効となります．CLEAR文はメモリ上の領域を設定するだけなので，メモリ上のデータやプログラムは，消えることなく保存されます．

CLEAR文ではデータなどは消えない

2 8086の特徴

前節ではPC-9801の構成について触れましたが，ここではその中心ともいえるCPU8086の特徴について述べることにします．本書の目的は，PC-9801を通して8086の機械語を理解することにあるので，この8086の基本的な特徴について理解しておくことは重要です．

1 16ビットCPUの生立ち

CPU
MPU

世界で最初につくられたマイクロコンピュータは，1971年にインテル社が発表した4004という4ビットCPU（Central Processing Unit—中央処理装置．MPU：Micro Processing Unit—マイクロプロセッサと呼ばれることもある）です．同社はさらに8ビットCPU8008を発表し，その後に続く8ビット時代の幕を切ったわけです．しかし，この8008は処理速度に問題があり，まだ十分とはいえませんでした．本格的な8ビットCPUの誕生は，1973年に発表された8080であるといわれています．8080は各方面に広く普及し，現在の80系CPUの先祖ともいえるものです．

その後，1976年には8080を改良した8085Aが発表され，さらにザイログ社から8080を元に，機能をより充実させたZ80が発表されるにいたって，これらのCPUをとう載したマイクロコンピュータが一せいに開発され，8ビットマイコンは広く普及し始めました．

80系のCPUは，上位機種が常に下位機種の機能や命令を含み，互換性を保つ形で開発がされてきましたが，インテル社はマイクロコンピュータの高性能化という要求に対応して，1978年に16ビットCPU 8086と8088を発表しました．これらのCPUは，8ビットCPUとの共通点を保ち続けた上で大幅な機能の拡充を行なうという同社の方針に従って開発されたものであり，いままでに8ビ

ットCPUで蓄積されたソフトウェアを，わずかに変更するだけで使用できるという利点を持っています．

ところで，マイクロプロセッサにはもうひとつの流れがあります．それはモトローラ社によって開発された6809などの68系と呼ばれるCPUです．

モトローラ社は，1974年に8ビットMPU（マイクロプロセッサ）6800を発表しました．そして1979年にはそれを改良したMPU6809を発表し，「究極の8ビット」としての地位を確立しました．確かに6809は優れたCPUであり，性能や使いやすさという点では8085やZ80より優れていますが，残念ながらその発表が80系に比べて遅れたこともあり，現在の普及率はいま一歩80系に及びません．

80系と68系の違いは，その設計思想にもあります．80系のCPUは，制御用や電卓用の素子を1チップ化するところから開発が始まったわけですが，68系のCPUは，汎用コンピュータのアーキテクチャにならってそれをマイクロコンピュータ化することを目標に開発されました．

どちらが優れているということは一概にはいえませんが，普及率では80系が，性能の点では68系がそれぞれリードしているといえます．

こうした違いは16ビットCPUの時代になっても受け継がれています．インテル社は，8085の機能を拡張する形で16ビットCPUである8086を開発しましたが，対するモトローラ社は，有名なDEC社のミニ・コンピュータPDP-11などのアーキテクチャにならって16ビットCPU68000を開発，発表しました．

8086が8ビットと16ビットの中間に位置するCPUであるのに比べ，68000は16ビットと32ビットの中間に位置するCPUであることから，性能的には68000の方が優れているわけですが，現状では8086の方が広く普及しています．日本で発売されている16ビットマイクロコンピュータも，そのほとんどが8086を使用したものとなっています．

現在，68000をとう載し，UNIXなどのミニコンレベルのOSを走らせることができるマイコンの開発も進んでいますが，一般に普及するまでには，まだしばらくかかりそうです．したがって，価格や普及率の点からみて，今後も8086が広範に使用され続けることは確かでしょう．8086を学ぶということは，そういった点からも重要であるわけです．

2 8086のレジスタ構成

それでは次に，8086のアーキテクチャについて述べていきましょう．まず最初は，レジスタの構成です．

CPUを構成する基本的な要素としてレジスタがあるということは，先に述べたとおりです．レジスタはメモリの一種であるわけですが，CPUの中にあってそれぞれが特殊な役割を担っているという点で，一般のメモリとは異なります．しかも，同じ演算をする場合でも，CPUの中にあることから，メモリを使用するより高速に演算が行なえるという利点を持っています．したがってレジスタの数や構成が，CPUの性能を決定する重要な要因となります．

汎用レジスタ

8086のレジスタの第1の特徴は，ある程度レジスタに汎用性をもたせたこと，いいかえればどのレジスタを使用しても同じように転送や演算を行なえるようにしたことです．この点は，8085やZ80のレジスタが，汎用性が低く，転送や演算関係の命令がほとんどAレジスタ1本に集中していたことを考えると，非常に進歩した点といえるでしょう．

独自の機能を持ったレジスタ

第2の特徴は，レジスタそれぞれに独自の機能を持たせることにより，定式化した命令を非常にコンパクトにまとめることができるようになった点です．たとえば，CXレジスタをカウンタとして繰り返しの数の計算に使用することにより，カウントと条件分岐とを一度に行なうことのできるLOOP命令などが使用できます．

以上あげた2つの特徴は，実は互いに矛盾する側面をも含んでいるため，両方の特徴を同時に活かすことは困難です．しかし，いずれにしても8ビットのCPUと比べると，進歩していることは確かです．

次に，8086のレジスタ構成と，それぞれのレジスタの機能について説明していきます．

図1.6　8086のレジスタ構成

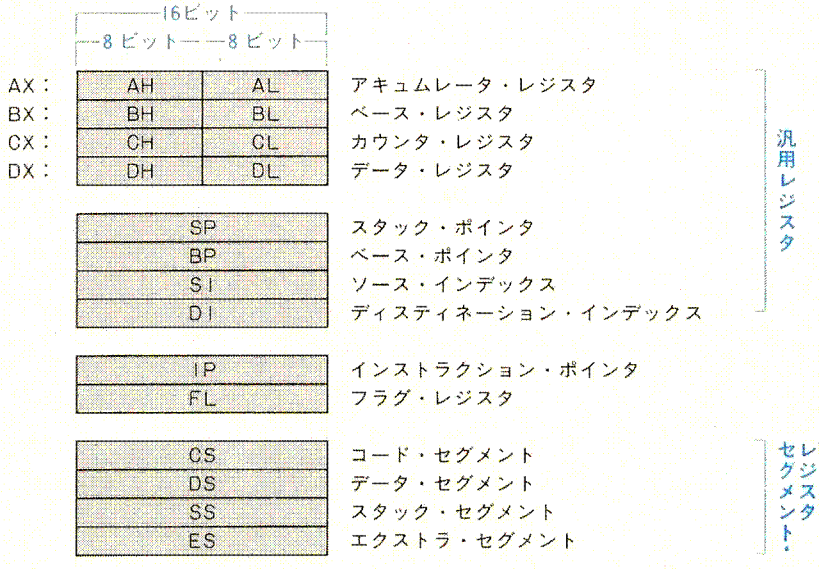


図1.6に示すように，すべてのレジスタは16ビットで構成されていますが，AX，BX，CX，DXの各レジスタは，それぞれ上位8ビット，下位8ビットに分けて使用することもできます（HiとLoとに分けて，AH，ALというように呼ぶ）．

AX, BX, CX, DXは8ビット，16ビットのどちらにも使える

汎用レジスタは，転送命令はもちろん，各種演算命令に用いることができますが，それらのうちAX，BX，CX，DXの4つのレジスタは演算を行なうために，SP, BP, SI, DIの4つのレジスタはアドレスを指定するために，主として使用されます．

次のIP（インストラクション・ポインタ）は，次に実行する命令のあるアドレスを指示するレジスタです．また，フラグ・レジスタは，演算の結果によってセットされる部分と，CPUの動作を制御する部分とから成り，通常のレジスタとは異なった働きをします．

最後に，セグメント・レジスタと呼ばれる4つのレジスタ（CS，DS，SS，ES）は，8086に特有のレジスタです．8086では1Mバイトのメモリを64Kバイトごとに区切って使用しますが，その際の区切りを示すレジスタです（詳しくは事項で述べます）．

それでは，以上のレジスタの機能を次にまとめておきましょう．

8086の各レジスタの機能

AX：アキュムレータ・レジスタ……転送や演算専用のレジスタ．他の汎用レジスタを用いても，転送や演算を行なうことはできるが，AXレジスタを使用することにより機械語命令の長さを短くすることが可能になり，結果的に能率を向上させることができる．

BX：ベース・レジスタ……転送や演算の他に，アドレスの間接指定に用いることができる．

CX：カウンタ・レジスタ……転送や演算の他に繰り返し命令などで，回数のカウントに用いる．このレジスタを使用すると，一定回数繰り返す命令の作成が容易になる．

DX：データ・レジスタ……転送や演算の他に，一部の演算命令でAXレジスタの補助としてデータを扱うのに使用される．

SP：スタック・ポインタ……スタックの最後のデータのあるアドレスを示すレジスタ．

BP：ベース・ポインタ……スタック上のアドレスを指定するためのレジスタ．SPは，プログラムに従ってCPUが自動的に値を増減するので，ユーザが使用すると混乱するが，BPはユーザが自由に使用できる．

ＳＩ：ソース・インデックス……データ転送の際のアドレスを指定するためのレジスタ．特に，メモリからレジスタにデータを転送する際の転送元（ソース）のアドレスを指定するために使用すると有効である．

ＤＩ：ディスティネーション・インデックス……データを転送の際のアドレスを指定するためのレジスタ．特に，レジスタからメモリにデータを転送する際の転送先（ディスティネーション）のアドレスを指定するために使用すると有効である．

ＩＰ：インストラクション・ポインタ……次に実行する命令が格納されているアドレスを指定するためのレジスタ．命令の実行に伴いCPUが自動的に値を変化させる．

ＦＬ：フラグ・レジスタ……一般のレジスタと異なり，ひとつひとつのビットが意味を持ち，独立して働く．演算の結果に従って，オーバフローその他の状態を示すビットが，セット（１になる）またはリセット（０になる）される．またCPUが割り込みを受け付けるか否か，シングルステップモードで動作させるか否かなどのCPUを制御するスイッチのような働きを持つビットもある．

ＣＳ：コード・セグメント……CPUが実行すべき命令が格納されているセグメント（64Kバイトの区間）の始めを示すレジスタ．

ＤＳ：データ・セグメント……データを転送する際に，データを格納するために使用するセグメントの始めを示すレジスタ．

ＳＳ：スタック・セグメント……スタック動作(後述)を行なうためのセグメントの始めを示すレジスタ．

ＥＳ：エクストラ・セグメント……データ転送命令の一部で使用され，データの転送先となるセグメントの始めを示すレジスタ．

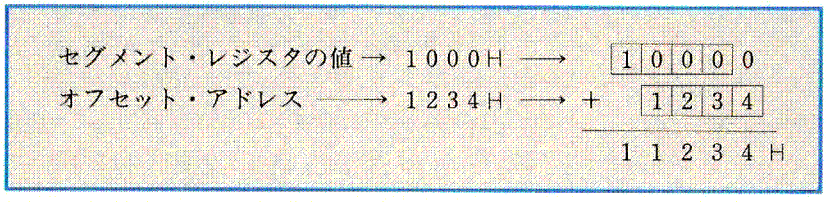
3 1Mバイトのメモリとセグメント

8086では１Mバイトまでのメモリが使用できることは，すでに述べたとおりです．１Mバイトのアドレスを指定するためには，16進数で0H～0FFFFFHまで，２進数で表わすと20ビットが必要となります．ところが，前項で述べたように，8086のレジスタはすべて16ビットになっています．16ビットでは，20ビットに

対して4ビットたりません．16進数では1桁分不足です．

そこで，次のような方法を用いてアドレスを指定します．すなわち，レジスタを2本使用し，片方が上から4桁を，もう片方が下から4桁を表わすものとして加え合わせます．

図で表わすと，下のようになります．



上から4桁を表わすために設けられたレジスタを，セグメント・レジスタと呼びます．上の図からわかるように，一番下の桁は0であるものとみなされて加算されます．一方，下の4桁をオフセット・アドレスと書きましたが，これは，**第3章**の「**アドレッシング・モード**」のところでも述べますが，レジスタの値である場合もあれば，メモリの内容または直接に与えられた数値である場合もあります．

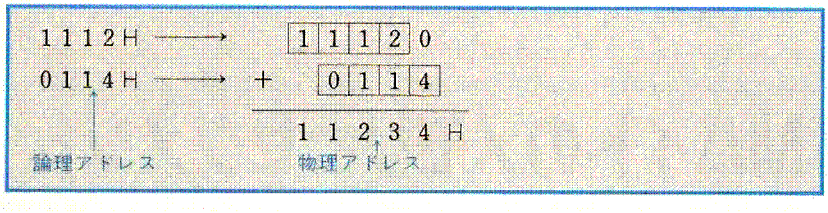
アドレスはセグメントを基本に計算する

オフセットとは，基準からのずれ，または変位のことです．つまり，セグメント・レジスタを基準にして，そこからのずれをオフセット・アドレスが示すと考えるわけです．

オフセットとはずれのこと

一般的な使用法では，はじめにセグメント・レジスタの値を設定しておき，それを基準としてオフセット値を変化させる形でアドレスを指定します．**こうして指定できるアドレスは、セグメント・レジスタの示すアドレスから，64Kバイトの範囲であり，この範囲をセグメントと呼びます．**

このように，実際のアドレスは2本のレジスタの組み合わせで求められるわけですから，同じアドレスを指定するのに，何通りもの方法があります．前の例と同じアドレスを指定するために，たとえば次のような表わし方もできるわけです．



こうして生成される16進5桁のアドレスを，物理アドレスと呼びます．物理アドレスは，メモリ上の場所に直接対応しています．一方，セグメント・レジスタの値やオフセット・アドレスのことを，論理アドレスと呼びます．

物理アドレスは，論理アドレスの組み合わせによって表わされます．上の例

のように，同じメモリ上のアドレスを表わすのに，物理アドレスでは1通りですが，論理アドレスでは何通りもの方法があります．

4 セグメントに分ける意義

ところで，アドレスを指定する際に，その都度2つのアドレスを指定しなければならないのでは，非常に面倒です．そこで，8086では，セグメント・アドレスの方は，一度指定してしまえば後は変更するまで同じ値を保つものとし，オフセット・アドレスのみを指定すれば物理アドレスが求められるようになっています（物理アドレスの計算は，CPUが自動的に行ないます）．

こうしておけば，セグメント・アドレスを一度指定してしまえばそこから64Kバイトの範囲内のアドレスは，オフセット・アドレスのみによって指定できることになります．実用上では，1Mバイトのアドレスを一度に使用することはあまりなく，64Kバイトもあればプログラムやデータを格納するためには十分ですから，このようなアドレスの指定法は便利な方法といえます．

4つのセグメントに分ける意味

それでは，何のためにセグメント・レジスタが4本あるのかというと，メモリ上を4つのセグメントに分け，それぞれ異なった目的に使用するためです．

たとえば，コード・セグメントは命令を格納するために使用します．CPUは，命令を取り出すときに，必ずコード・セグメントから取り出すことになっているので，プログラムはコード・セグメント内に書かなければなりません．

また，データ・セグメントは，データを格納するためのセグメントであり，8086の命令ではデータの操作は主としてデータ・セグメント内で行なわれます．

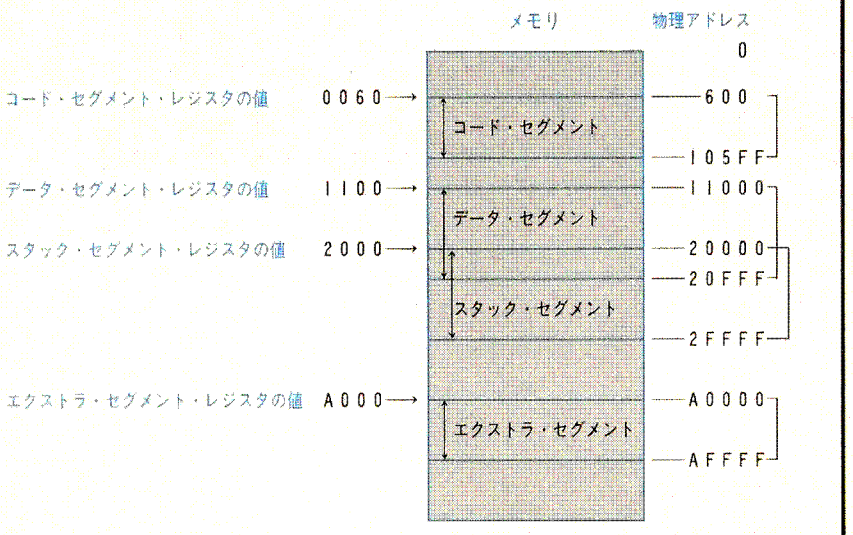
スタック・セグメントは，スタック動作（**第5章「サブルーチンとスタック」**のところで詳述する）を行なうためのメモリ領域であり，プログラムの流れを制御するためのデータなどが一時的に格納されるセグメントです．

エクストラ・セグメントは，データを格納するセグメントであるという点でデータ・セグメントと同じですが，こちらの方は予備であり，データ・セグメントの補助的役割に使用されます．

これらのセグメントの場所は，セグメント・レジスタの値によって示され，1Mバイトの範囲内のどこにでも置くことができます．また，領域が重なってもかまいません（その際は，プログラム作成時に注意が必要）．

物理アドレスの上にセグメントを割り付けることを，セグメンテーションといいます（**図1.7**参照）．セグメントの順序は，別に**図1.7**の例のとおりにする必要はありません．セグメント・レジスタの値を変更しながら使用することによって，1Mバイトのメモリをすべて使用することができますが，その際には

図1.7　メモリのセグメンテーションの例



十分な注意が必要です．

8ビットCPUには，このようなセグメンテーションの概念がなく，命令もデータもスタックも，すべてまとめて64Kバイト内に納めなければならなかったことを考えると，それらの領域を区別して使用できるこのセグメンテーションという方法は，16ビットCPUならではのメリットといえるでしょう．

5 CPUの内部構成

CPUの中ではどんなことが行なわれているか，ここで少しのぞいてみることにしましょう．

なお，ここで述べることはハードウェアに関連したことであり，マシン語を理解するために特に必要な事柄というわけではありません．したがって，手早くマシン語を習得したいという方は，読みとばして次の**2章**に進んでいただいて結構です．

次ページ**図1.8**は，8086CPUの内部ブロック図を示したものです．

この図は，CPUの内部構成を表わしたものですが，CPUは大きく分けると実行ユニット（Execation Unit）と，バス・インターフェイスユニット（Bus Interface Unit）とに分けられます．前者は演算処理を行なうユニットであり，後者はアドレスの計算を行なって命令やデータの相互転送を行なうユニットです．

EUとBIU

図1.8　8086CPU内部ブロック図

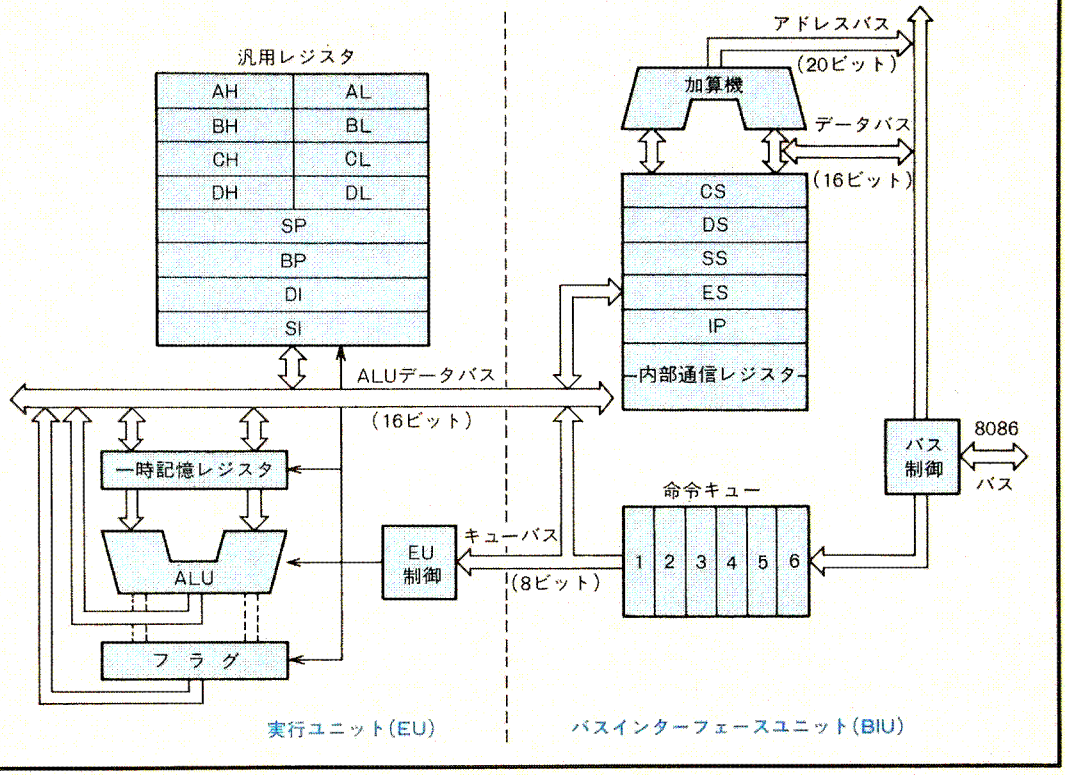


図1.8の中でALUと示されている部分は，算術演算ユニット（Arithmetic & Logic Unit）のことであり，CPUでの演算はすべてここで行なわれます．その他の部分は，このALUに送るデータを蓄えたり，制御したりする部分です．　ALU

ALUと同じような形をした部分で，加算機と呼ばれるものがありますが，これは先に述べたように，セグメント・レジスタとオフセット値から，物理アドレスを生成するためのユニットです．命令の実行中はCSとIPの値からアドレスを生成して命令を取り込み，データの転送時にはDS(または指定されたセグメント)と，オフセットアドレスからアドレスを生成してデータの転送先(元)を決定します．

命令キューと呼ばれる部分は8086に特徴的な部分であり，命令の先読みを行なって蓄えておく部分です．

CPUの動作としては，

① **命令の読み込み**

② **解釈・演算**

③ **データなどの読み込み・書き込み**

の3つがあります．

従来のCPUでは，①と③の場合にはバスを使用してメモリとデータの交換を

行ないますが，②のときにはバスを使用しません．そのとき，バスは遊んでいるわけです．

8086では，このバスの遊びの時間をなくし，実行ユニットが演算などを行なっている間に次の命令を読み出して，キューと呼ばれる一時的なメモリに蓄えておきます．このような動作を，プリ・フェッチ（フェッチとは取り込むという意味）と呼びます．このプリ・フェッチによって，バスを効率よく使用することができ，結果として実行速度が向上します．

図1.9　8086CPUと従来のCPUとの違い

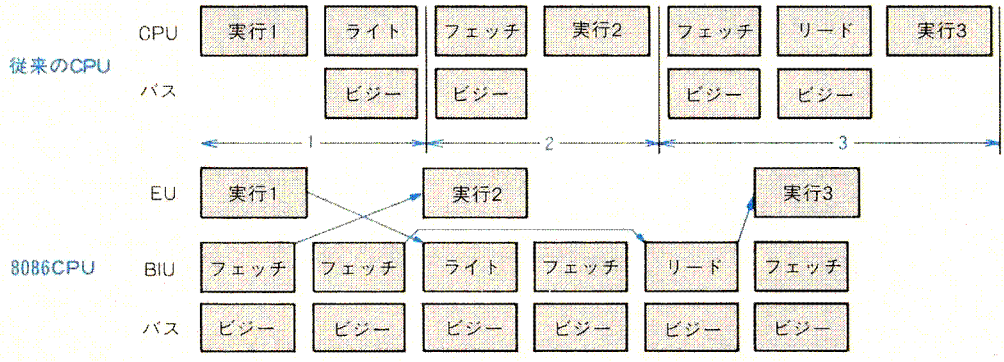


図1.9からわかるように，3つの命令（それぞれ処理の流れが異なる）を実行するまでにかかる時間が，8086では従来のCPUよりも少なくてすみます．さらにその上，2つも多くの命令がフェッチされています．

一般に，実行にかかる時間は，フェッチやリード・ライトにかかる時間よりも長くかかるので，実行時間中にバスが有効に使用できるということは，大きなメリットとなります．

命令の先読みで実行速度が向上

8086では，この命令キューが6バイト（8088では4バイト）あり，先に読み込んだ命令を先に出すという，先入れ先出し動作(FIFO：First In First Out)が行なわれています．

なお，このようなプリ・フェッチ動作は，アドレスの下位から上位に向かって行なわれていくので，プログラムが直線的に順番に並んでいるときは有効ですが，分岐命令などによって別のアドレスから新しく命令を実行する場合には，いままでにキューに取り込んだ命令はキャンセルされ，新しく命令を取り込まなければならなくなります．そのため，バスを有効に使用するためには，分岐命令を多用することは避けた方がよいのですが，実用上はあまり問題にはならないので，特に気を使わなくてもかまいません．

6 TTLの基礎知識

少し横道にそれますが，ここでTTL回路の基礎について触れておくことにします．8086を理解するために直接必要となる知識というわけではありませんが，回路図をみるときに必要となる程度の予備知識を持っておくのも悪くはないと思います．ただし，興味のない方は読み飛ばして，次の章に進まれてもかまいません．

TTL

TTLとは，Transistor Transistor Logicの略であり，トランジスタを用いて構成された論理回路のことです．この論理回路では，電圧のある・なしを，1と0とに対応させています．TTLで使用される電圧は一般に5Vであり，5Vがかかっているか否かで1か0かの状態を表わしているわけです．現在，ディジタル回路を構成するICやLSIは，この規格になっているものがほとんどです．0と1だけを用いて複雑な計算を行なうためには，後で述べる2進数を使用します．

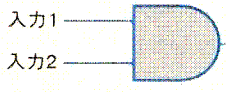
論理回路

さて，次は論理回路の基礎について説明しましょう．

ICは，入力された信号に基づいて判断を行ない，出力信号を出すわけですが，その際，信号が「真」か「偽」かに基づいて出力を決定します．ここでは，AND回路を例にとって説明することにします．

AND回路は，2本の入力信号のANDをとって出力する回路です．つまり，両方とも「真」のときだけ出力を「真」にする回路です．入力に対する出力の真偽を示した表を真理値表と呼びますが，**図1.10**は，AND回路とその真理値表を示したものです．

図1.10　AND回路

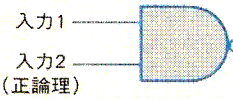


ANDの真理値表

入力1	入力2	出力
真	真	真
真	偽	偽
偽	真	偽
偽	偽	偽

TTLレベルで真偽を表わすときに，0Vを偽，5Vを真とみなすような方法を**正論理**と呼びます．**図1.10**の回路は正論理のAND回路ですから，入力1，2とも5Vのときだけ出力が5Vになり，それ以外のときは0Vになります．

図1.11



また，0Vを真，5Vを偽とみなすような方法を，**負論理**と呼びます．

正論理・負論理

実際には，TTL回路では正論理と負論理を組み合わせた回路が多く使用されます．たとえば，**図1.11**に示すようなAND回路がそうです．

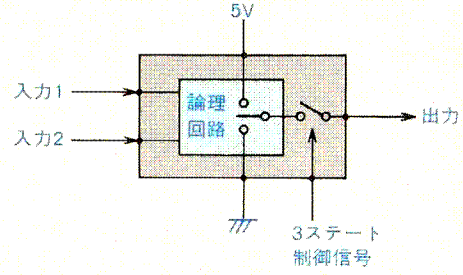
この回路では，入力1，2ともに真，すなわち5Vのときだけ出力が真，つまり0Vになり，それ以外の場合は5Vになります．

なお，図に示したように負論理の入出力端子には「○」を付けて表わします．

したがって，「○」の付いた端子は0Vのときが真となります．ICの端子配置図において，ピンに「○」が付いているものとそうでないものとがあるのは，5Vを真とみなす端子であるのか，0Vを真とみなす端子であるのかを区別しているのです．

ところで，こうした論理回路の出力端子電圧は，0Vまたは5V（多少の変動は許される）となるわけですが，その他に，第3の状態として，ハイ・インピーダンスと呼ばれる状態をとることができるものがあります．この状態は，論理回路の中でどこにも接続されていない状態と等価であり，接続を断つことにより他の論理回路に影響を与えなくなるので，1本の信号線にいくつもの論理回路の出力が接続されている場合には有効です．この状態を**3ステート**と呼び，別の信号線によってコントロールされます．

図1.12　3ステートの模式図



また，3ステートのICは，単なるスイッチとして使用されることもあります．後で説明するように，バス上にはアドレス情報とデータとが，時間を分割して現われますが，このバスからアドレス情報のみを取り出すために，アドレス情報が表われたときだけ「ON」になり，他のときは「OFF」つまり3ステート状態になるようなICが使われます．そのために，アドレス情報が送られているか否かを示す別の信号線があり，このICはその信号線によってコントロールされます．

3ステート

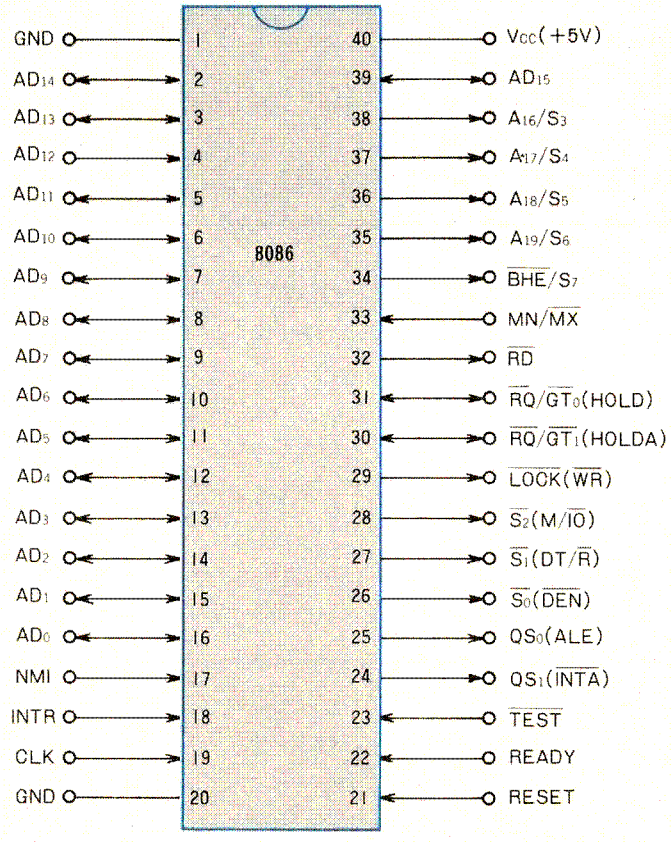
7 8086の端子の働き

次ページ図1.13は，上面からみた8086の端子接続図です．

図中の矢印は，信号の方向を示しています．V_{cc}は電源用の5Vの入力であり，GNDはアース端子です．なお，CPUのような複雑なLSIでは，正論理か負論理かを一概には決められないので，端子には特に「○」印は付けてありません．

（ ）内は，ミニマムモード時の信号を意味しています．PC-9801では，8086をマキシマムモードで使用しているので，（ ）内は関係ありません．

図1.13　8086端子接続図



それでは以下に，各端子信号の機能についてまとめておきましょう．

AD15～AD0(Address and Data)●3ステート入出力

これらのラインは，アドレスの出力とデータの入出力に使います．同じラインを使用して，アドレスとデータの両方を扱うために，時間的に切換えて（時分割で）使用しています．どちらを扱う時間かということは，バス・コントローラから出力されるALE(Address Latch Enable)信号によって決定されます．

CPUは，ごく短時間の間にこのラインにアドレス情報を出力し，データ情報を入力（出力）するという操作を繰り返します．

A19/S6～A16/S3(Address/Status)●3ステート出力

アドレスを示すためには20ビットの情報，いいかえれば20本のラインが必要です．上記の16ビットと併せて，20ビットのアドレス情報を出力します．また，時分割によりアドレスが使用していないときには，ステータス情報を出力します．ステータス情報は，それぞれ**表1.1** に示すようになっています．

表1.1　ステータス情報

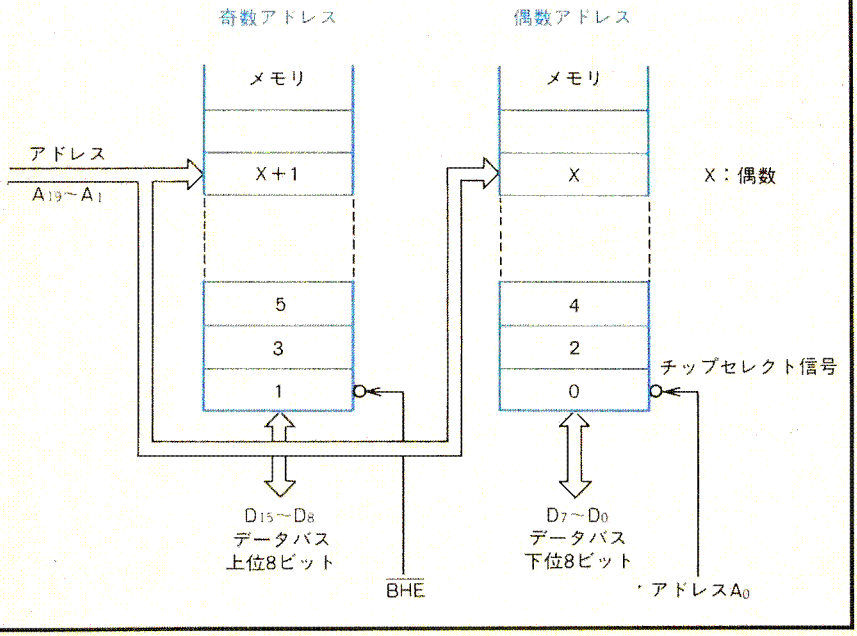
S6：常に0 S5：割り込みイネーブル・フラグの状態（0のとき割り込み禁止）		
S4	S3	データ・アクセスに使用されるセグメント・レジスタ
0	0	ES
0	1	SS
1	0	CSまたはセグメントなし（I/Oのアクセス時など）
1	1	DS

BHE/S7(Bus High Enable/Status)●3ステート出力

8086には，データバスが16本あり，一度に16ビットのデータを扱うことができます．ところが，前節で述べたように，メモリは8ビットを単位として構成されています．それでは，どうやって16ビットのデータを一度に読み出すことができるのでしょうか．

実は，8086を使用したシステムでは，偶数アドレスのメモリと奇数アドレスのメモリが並列に配置されています(図1.14参照)．そして，偶数アドレスのメモリがデータバスの下位8ビットに，奇数アドレスのメモリがデータバスの上位8ビットに接続されています．

図1.14



このように接続した上で，偶数バイトと奇数バイトを同時に指定すれば，一度に16ビットのデータがデータバス上に得られるわけです．

さて，その方法ですが，まず図1.14のように偶数アドレスのメモリと奇数ア

ドレスのメモリを並べて配置し，アドレス信号が偶数アドレスのXを指しているときには，同時に奇数アドレスのX＋1も指すようにしておきます．

偶数アドレスを指定したときにはA0は0になるので，偶数アドレスのメモリのチップセレクト信号がLO（ロー）になります．つまり，このチップのデータが有効となるわけであり，偶数アドレスXのデータがD_7～D_0に得られます．さらにそのとき，$\overline{BHE}$信号をLOにすると，奇数アドレスX＋1のデータが同時にD_{15}～D_8に得られるので，一度に16ビットのデータを取り出すことができるわけです．おわかりいただけたでしょうか．$\overline{BHE}$（バス・ハイ・イネーブル）とは，バスの上位が有効（可能）であるという意味です．

イネーブル

なお，$\overline{BHE}$のように，名称の上に ‾ （バー）の付いている信号は，ロー・アクティブ，すなわちLO状態（0V）のときに意味を持つという信号です．

ところで，8086では16ビットのデータ転送ばかりではなく，8ビット単位でのデータ転送もできます．そのときは，次のようにします．

① **偶数アドレスのデータをアクセスする場合** ― 偶数アドレスを指定し，$\overline{BHE}$信号をHI（ハイ）にすると，奇数アドレスはセレクトされず，偶数アドレスのデータのみがD_7～D_0に現われる．

② **奇数アドレスのデータをアクセスする場合** ― 奇数アドレスを指定し，$\overline{BHE}$信号をLOにする．A_0＝1なので偶数アドレスは選択されず，$\overline{BHE}$＝LOなので奇数アドレスのメモリのデータのみがD_{15}～D_8に現われる．

なお，Xを偶数としたときに，X＋1，X＋2という形で16ビットのデータを得るためには，2回に分けて転送しなければならないため，時間が余分にかかります．8086で16ビットのデータを扱うときには，常に偶数アドレスが下位になるようにしないと，効率が悪くなります．ここでは，データの読み出しについてのみ触れましたが，書き込みの場合もまったく同様です．

これらの信号は，当然，命令に従ってCPUが自動的に制御してくれるので，使用者はあまり気にする必要はありません．

説明が長くなりましたが，$\overline{BHE}$信号は，このように奇数アドレスのメモリ・セレクトを決める信号です．なお，S7は現在使用されていません．

MN/MX●入力

8086をミニマムモードで使用するか，マキシマムモードで使用するかを決める信号です．このような書き方をした場合，HIレベルであればMN（ミニマム）が，LOレベルであれば$\overline{MX}$（マキシマム）が選択されることを意味します．

ミニマムモード
マキシマムモード

ミニマムモードの場合，バスのコントロール信号はすべてCPUから出されるため，ひとつのCPUのみで構成されたシステムを作るときに使用します．

一方，マキシマムモードの場合は，バスのコントロールのための状態情報$\overline{S0}$～$\overline{S2}$がCPUからバス・コントローラに供給され，バスのコントロールはこの

バス・コントローラによって行なわれます．いくつものLSIがバスを共有して使用するような大規模なシステムに使用されます．なお，PC-9801ではマキシマムモードが使用されています．

RD(Read Strobe)● 3ステート出力

この信号がLO，すなわちアクティブであるとき，メモリに対する動作がリード動作であることを示しています．

RQ/GT0, RQ/GT1(Request/Grant)● 入出力

これらの信号は，他のローカル・バス・マスタ（コ・プロセッサやDMAコントローラなどの周辺コントローラで，バスを使用するもの）が，CPUを介さずにデータの転送を行なう際に，CPUに対してバスの使用権を要求する信号です．

CPUは，この信号を受けると現在の処理を終了後，同じ信号線に信号を出してバスの使用権を許可します．周辺コントローラは，バスの使用が終了すると，再び同じ信号線に信号を出し，CPUにバスの使用権を返します．GT 0の方が，GT 1よりも優先されます．

LOCK(Lock)● 3ステート出力

CPUがLOCK命令を実行すると，その次の命令を実行する間，バスの使用権を確保し，他のプロセッサにバスの使用権を与えないことを示す信号です．この信号がアクティブの間，他のプロセッサのバスの使用要求は無視されます．

S2, S1, S0● 3ステート出力

この3本の信号は，ミニマムモードかマキシマムモードかによって，若干その働きが異なります．PC-9801では，8086はマキシマムモードで使用されているので，ここではそのときの働きについて述べておきます．

マキシマムモードでは，この3本の信号は組み合わせて使用され，**表1.2**に示すような動作状態であることを意味します．

表1.2

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	
0	0	0	割り込み応答
0	0	1	リードI/O
0	1	0	ライトI/O
0	1	1	ホールト
1	0	0	命令コードアクセス
1	0	1	リードメモリ
1	1	0	ライトメモリ
1	1	1	受動状態(ノンバスサイクル)

これらの3本の信号線はバスコントローラと呼ばれるLSIの入力につながっていて，このLSIから表の右に示したような信号が発生するようになっています．バス・コントローラは，$\overline{S2}$～$\overline{S0}$信号の発生するタイミングをもとに，バスの信号がアドレスであるかデータであるかを示す信号(ALE)やメモリ，I/O選択信号，リード・ライト選択信号などを正しいタイミングで発生させます．

QS0, QS1(Queue Status)●出力

この2つの信号線は，8086内部の命令キュー（前述）の状態を示しています．

表1.3

QS1	QS0	
0	0	キューの変動がなかったことを示す
0	1	キューから命令の1バイト目が取り出されたことを示す
1	0	キュー内が空であることを示す
1	1	キューから命令の2バイト目以降が取り出されたことを示す

TEST(Test)●入力

CPUはWAIT命令を実行すると，この$\overline{\text{TEST}}$信号が入力されるまでWAIT状態となります．CPUが，外部からの入力を待って動作を始めるときに使用しますが，主として8087数値データプロセッサの演算結果を待つときに使用されます．

READY(Ready)●入力

メモリや入出力装置などが，CPUからの要求に応じてデータの入出力を行なう準備ができたことを示す信号です．CPUは，アドレスを指定した後，メモリなどからREADY信号が送られてくるのを待って，データの入出力を行ないます．

RESET(Reset)●入力

CPUのすべての動作を終了させ，システムを再スタートさせるための信号です．この信号を受け付けると，CPUはすべての動作を中止し，新しいアドレスとしてコードやセグメントに0FFFFHを，IP(インストラクションポインタ)に0Hを設定し，そこから実行を開始します．つまり，物理アドレスの0FFFF0Hから新しく実行が開始されるわけです．

NMI(Non Maskable Interrupt)●入力

マスクすることのできない外部割り込み要求信号です．この信号を受け付けると，タイプ2の割り込み(後述)を発生して，割り込み処理ルーチンに移ります．

INTR(Interrupt Request)●入力

外部機器からの割り込み要求信号です．IF（インタラプト・イネーブル・フラグ）が1のときのみ，この信号が受け入れられます．この信号は通常，割り込みコントローラと呼ばれるICを通して発生され，CPUはこれを受け付けると

割り込み応答信号（$\overline{S2}$～$\overline{S0}$がすべて0）を発生し，割り込みコントローラによって指定されたタイプの割り込みベクトルを参照して，割り込み処理ルーチンに処理を移します（割り込み動作については後で説明する）．

CLK(Clock)●入力

CPUその他の周辺機器が，同期をとって動作するためのクロック信号の入力です．クロックは専用のICによって発生され，すべてのIC，LSIに供給されます．PC-9801では5 MHz，PC-9801E/Fでは8 MHzがクロック周波数であり，CPUとメモリ間のデータ転送には4クロック以上（5MHzで800 ns：ナノ秒＝10億分の1秒），演算には数クロック～百数十クロックを必要とします．クロックに従って，CPUは様々な信号線のコントロール情報の入出力を行ない，演算やデータの転送などを行ないます．

以上，各端子信号について説明しましたが，これらのことは，これからハードウェアについても勉強を始めようという人のための解説ですから，マシン語の入門にはあまり必要のないことであったかもしれません．いつかまた，必要が生じたときに読みかえしていただくことにして，理解できなくてもかまいませんから，次に進むことにしましょう．

第2章

マシン語の基礎知識

1　2進数と16進数

2　機械語とアセンブリ言語

3　モニタの使い方

1 2進数と16進数

1 10進数と2進数

私達は日常，数を数えるときに10進法を用います．これは，おそらく物の数を勘定するときに，両手の指を使ったからでしょう．

この10進法では，数を1，2，3，4……と数えていき，9まできたらひとつ桁をあげて10というように数えていきます．これを，数学的に表現して，123という数値が10の累乗を使って，

$$1 \times 10^2 + 2 \times 10 + 3$$

と表わされる数値である場合，これは10進数であり，このような定義に基づいた体系を10進法と呼んでいます．

ところで，コンピュータが計算を行なうときはどうでしょうか．

なぜ2進数を用いるのか

周知のとおり，コンピュータは電気を使って計算を行なっているわけですが，たとえばそこで10進数を使用するとすれば，0～9までの数を電圧に対応させて，0V，1V，……9Vというようにしなければなりません．しかし，この方法には問題があります．というのも，数値に従って0V～9Vの電圧を発生させるとすれば，そのための装置は複雑なものとなるし，何よりノイズや電圧の変動によって，9Vが8Vになってしまうといったこともあり得ます．

そこで，10進数の代わりに2進数という考え方を導入します．2進数では，用いる数は0と1の2つだけです．したがって，対応させる電圧も"ある"か"ない"かということだけがわかればよいわけです．極端にいえば，スイッチのON，OFFによって数値を表わすことができることになり，装置も簡単で，ノイズや電圧の変動にも強いことになります．

それでは次に，2進数を使用して，どのように数を表わすのかということについて説明しましょう．

2進法の基本は2の累乗

まず，上で述べた，10進数を用いた数値の表現の例を，2進数についても適用します．

たとえば，111という2進数の数値は，

$$1\times2^2+1\times2+1$$

という数のことですから，つまり$1\times4+1\times2+1=4+2+1=7$ということです．

このように2の累乗を基本にして数値を表現する方法を，2進法と呼びます．

2進数を10進数に直すには，上に述べたような方法を用いればよいのですが，10進数を2進数に直すには次のようにします．

10進数 → 2進数への変換

たとえば，10進数の123を2進数に直す方法を考えます．2の累乗を順に並べていくと，

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
1	2	4	8	16	32	64	128

となります．したがって，123は2^7より小さいことがわかります．そこで，2^6以下，次のように順に減算を行なっていきます．

元の数 →	123−64×1 →59	2^6の位……1
	59−32×1 →27	2^5の位……1
	27−16×1 →11	2^4の位……1
	11− 8×1 → 3	2^3の位……1
	3 − 4×0 → 3	2^2の位……0
	3 − 2×1 → 1	2^1の位……1
	1 ——→ 1	2^0の位……1

これで，10進数の123は2進数で表わすと，1111011であるということがわかります．ただし，数字だけを書くと10進数なのか2進数なのかわからなくなってしまうので，数値の最後に2進数であることを表わすための"B"(Binary)を付けて，

1111011B　または　1111011b

というように書きます（Bは小文字のbにすることもある）．

10進数の数値には，10進数であることを表わすための"D"(Decimal)を付けて，

123D　または　123d

と書きます．

後で述べる16進数と混乱しないためには小文字の方がよいかもしれませんが，大文字でも間違えることはないと思います．

2

2進数の加減算

以上で，コンピュータが2進法を使用して計算するということ，また2進法のあらましについて，大体のことはおわかりいただけたことと思います．ここでは，具体的には2進数を使ってどのように計算を行なうのか，実際に加減算を行なってみましょう．

2進数の加減算といっても，特別難しい計算ではありません．私達が普通に10進数の計算を行なう場合とまったく同じようにすればよいのです．ただし，注意しなければならないのは，使用できる数字が0と1だけであるということです．したがって，1＋1は2ではなく，1桁繰り上がって10となります．

試しにやってみましょう．

```
桁上がり →  1 1 1 1             1
           -----------
            1 0 1 1 0 1   ←    4 5
       +    1 0 0 1 1 1   ←   +3 9
      -------------------    ------
          1 0 1 0 1 0 0        8 4
```

下から3桁目の加算のとき，1＋1＋1 → 11になることに注意すれば，あとは簡単です．

ひき算もやってみましょう

```
      借り →   1010
           -----------
            1 0 1 1 0 1   ←    4 5
       -    1 0 0 1 1 1   ←   -3 9
      -------------------    ------
            0 0 0 1 1 0          6
```

上の桁から借りると，上の桁は0となってしまいます．上の桁から借りてきて，10－1 →1となることさえわかればよいわけです．10進数の加減算よりもよほど簡単です．10個も数字を使う10進数に比べて，2進数は2個しか数字がなく，しかも計算のルールが同じわけですから，2進数の方が簡単なのは当然といえます．ただ，私達が2進数の計算に馴れていないので，難しく感じるだけです．

3 レジスタと2進数

3-1　レジスタのビット数と2進数

前章の8086のレジスタ構成を説明するとき，みなさんが2進数について一応は理解しているという前提のもとに話を進めましたが，ここでもう一度レジスタのビット数について述べておきたいと思います．

レジスタが数値を記憶するためのものであり，その単位が8ビットとか16ビットとかであるということは，先に述べたとおりです．1ビットとは，2進数の1桁に対応するものであり，2進数8桁を記憶することのできるレジスタを8ビットレジスタ，16桁を記憶することのできるレジスタを16ビットレジスタといいます．したがって，8ビットレジスタでは，0B～11111111Bまでの数値を扱うことができます．10進数でいうと，0D～255Dまでです．また，16ビットレジスタでは，0B～1111111111111111Bまで，つまり0D～65535Dまでの数値を扱うことができるわけです．コンピュータでは普通，8ビットを単位としてこれを1バイトと呼びます．したがって16ビットは2バイトになります．2バイトのことを1ワードともいいます．

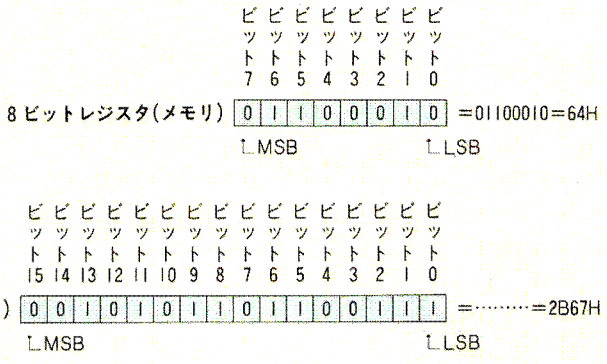
ビット
バイト
ワード

3-2　2進数のレジスタ内表現

レジスタやメモリが2進数を記憶するときには，たとえば8ビットのレジスタでは，8つのビットのそれぞれが0または1の値をとり，2進数を表現します．また，レジスタの中のビットを，2進数に対応させて，最上位の桁をビット7，次の桁をビット6というように順にビット0まで番号を付けて表わします(8ビットレジスタの場合)．なお，8ビットレジスタ，16ビットレジスタともに，最上位のビットを特にMSB(Most Significant Bit)，最下位のビットをLSB(Least Significant Bit)と呼びます．

MSB
LSB

図2.1



4 2進数と16進数

ところで，2進数はコンピュータにとっては都合のよいものですが，人間にとっては大変扱いにくいものです．

たとえば，

10進数の100Dは，2進数では01100100 B

10進数の11111Dは，2進数では0010101101100111 B

となりますから，ちょっとみただけでは何のことか，さっぱりわかりません．また，覚えたり書いたりするにも大変です．

そこで，この2進数をもう少し人間にわかりやすくするために，16進数という概念を導入します．なぜ“16進”かというと，$16=2^4$であり，2進数の4桁を16進数の1桁に対応させることができるからです．

16進数で数字を表わすためには，10進数で10個の数字が必要であったように，16個の数字が必要となります．そこで，10進数において普通に使用している0～9の数字に加えて，A～Fまでの文字を数字とみなして使用します．

10進数と16進数の対応は，次のようになります．

10進数	0	1	2	……	8	9	10	11	12	13	14	15	16
16進数	0	1	2	……	8	9	A	B	C	D	E	F	10

上のように対応させると，0～15までの数値を0～Fまでの1桁の数値で表わすことができます．

16になって，はじめて2桁になり，16進数で10となります．したがって，たとえば16進数の123は，16の累乗を用いて表現され，

$$1\times16^2+2\times16^1+3=291(10進数)$$

となります．

それではここで10進数と2進数と16進数の対応表を表2.1に示しておきます．

表をみるとわかるように，2進数の4桁が16進数の1桁に対応しています．

ここで，先の2進数を16進数に置き換えてみましょう．

10進数	2進数	16進数
100	0110 0100 B	64 H
11111	0010 1011 0110 0111 B	2B67 H

表2.1　10進数・2進数・16進数の対応

10進数	2進数	16進数
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10

このように，2進数を4桁ずつに区切って，対応表に従って16進数に直せばよいわけです．2進数の長たらしい表現が，16進数の短い表現となってすっきりとしました．なお，16進数の末尾には，16進数であるということを示す"H"（Hexadecimal）の記号を付けます．

16進数には"H"を一番上に"0"を付ける

さらに，変数名との混乱を避けるために，一番上の桁がアルファベットで始まるときには，その前に0を付けます．すなわち，

16進数の	1234	→	1234H
〃	A012	→	0A012H
〃	FF	→	0FFH

というように書きます．一番上の0はダミーですが，アルファベットで始まる数値のときには，必ず付けるようにしてください．

5 負の数と補数表現

ここまで扱ってきた数値は，すべて正の数でした．それでは，負の数を表現するときにはどうするのでしょうか．

私達が普通に数字で表現するときには，数字の前に"＋"または"－"の符号を付けて正負を表わしています．また，符号が省略されている場合は正の数を表わすということは，御存知のとおりです．

ここで，コンピュータの扱う2進数の場合を考えてみましょう．

もし，通常の数値と同じように符号を付けて表わすとすると，上で説明したレジスタまたはメモリの他に，符号を表わすためのレジスタまたはメモリが必要となります．

もちろん，こうした方法でも負数を表現することはできますが，符号とデータを別々に扱うことの問題などもあり，一般には使用されていません．

補　数

そこで，8ビット（または16ビット）のメモリだけで数値と符号の両方を表わせるように，と考えられたのが，負の数を**補数**を用いて表現する方法です．

補数とは，基準の数からある数を減じた残りの数のことです．

aを基準にとって，bの補数を求めるとき，bの補数を$\overline{b}$で表わすと，

$$\overline{b} = a - b$$

となります．

また，10進数で10を基準にとった3の補数を求めると，

$$3\text{の補数} = 10 - 3 = 7$$

となります．

これを，2進数にも適用します．8ビットの2進数の補数を求めるときには，

次のように2^8を基準にとって減算した結果を求めます．

たとえば，1に対する補数は，

$$\begin{array}{rr} 2^8 \rightarrow & 100000000 \\ - & 00000001 \\ \hline & 11111111 \end{array}$$

したがって，00000001の補数は，11111111となるわけです．これを16進数に直していうと，01Hの補数は0FFHとなります．この2つをたすと，01H＋0FFH→100Hとなるので，正しいことがわかります．

16ビットの場合も同様であり，2^{16}を基準にとって3の補数を求めると，

$$\begin{array}{rr} 2^{16} \rightarrow & 10000000000000000 \\ - & 0000000000000011 \\ \hline & 1111111111111101 \end{array}$$

したがって，これを16進数で表わすと，0003Hの補数が0FFFDHであることがわかります．

このような方法を，一般に**2の補数をとる**といいます（正確には2^8の補数または2^{16}の補数）．

2の補数

さて，このようにして補数をとることにより，負の数を表現することができるわけですが，そのとき困ったことが起こります．たとえば，0FFHは－1Hのことですが，はたして0FEHは－2Hのことだと考えてよいのでしょうか．負の数は，いくつまで考えられるのでしょう．

極端にいえば，80Hは，正の数の80Hとも，負の数の－80Hとも考えられるわけです．

そこでひとつの約束として，2進数で表現したときに最上位ビット（MSB）が1となっているものを負の数であると定義します．そうすると，たとえば8ビットの数値では0H～7FHは正の数，80H～0FFHまでは－80H～－1Hまでの負の数を表わすことになります（80Hより大きな数は，最上位ビットが1になっている）．16ビットでは，0H～7FFFHまでが正の数，8000H～0FFFFHまでは負の数となります．

MSBが1なら負の数

補数表現を用いると便利な点は，正負の数を区別せずに加算ができることにあります．

たとえば，1（01H）と－1（0FFH）を加えると，0になります（01H＋0FFH → 100H：下2桁のみ有効なので0H）．また，1(01H)と－2(0FEH)を加えると－1になります(01H＋0FEH→0FFH)．

数学的な証明は専門書にゆずりますが，この方法により，定められた範囲内で正負の数の正しい演算結果が得られます．

なお，「補数表示を用いて－8000Ｈ～7FFFＨまでの正負の数を表現する」のか，「補数表示を使わずに０Ｈ～0FFFFＨまでの正の数を表現する」のかは，計算をしたり，プログラムをつくったりする人の決めることであり，コンピュータにはそれを判断する能力はありません．したがって，プログラムをつくる人は，はじめから補数表示を用いるのか否かをきちんと決めて，プログラムをつくる必要があります．

表2.2　2の補数（８ビットの場合）

2　進　数 ↑MSB	16　進　数 （補数を用いない場合）	16　進　数 （補数を用いた場合）	補数と考えた時
00000000	0	0	
00000001	1	1	
00000010	2	2	正
↓	↓	↓	数
01111110	7E	7E	
01111111	7F	7F	
10000000	80	－80	
10000001	81	－7F	
10000010	82	－7E	負
↓	↓	↓	数
11111110	FE	－2	
11111111	FF	－1	

※便宜的に０は正の数として扱う．

※補数を求めるには，16進数の減算を用いてもよいが，２進数で表わされた数値の各ビットを反転させた上で１を加えてもよい．

例　－7Eを求める．　　正の数7E⇨01111110

⇩

ビット反転10000001

⇩

１を加える10000010＝－7E(負の数)

※補数表示で表わされた負の数を正の数に直すときも同様にすればよい（反転させて１を加える）．

2 機械語とアセンブリ言語

1 人間と機械の橋渡し・アセンブリ言語

マイクロ・コンピュータに仕事をさせるには，まずプログラムをつくらなければなりません．そのために，みなさんはBASICを使用するときはBASIC言語で，PASCALを使用するときにはPASCAL言語で，というように，ある文法規則に従った言語でプログラムを書くはずです．しかし，CPUはそれらの言語を直接理解することはできませんから，ある規則に従って機械語に翻訳してから，実行することになります．これらの翻訳をするためのシステムのことを，インタープリタとかコンパイラなどと呼びます．

インタープリタ
コンパイラ

インタープリタとは，プログラムを1行ごとに機械語に翻訳して実行するシステムのことであり，BASICでは通常この方法がとられています．コンパイラは，一度に全部のプログラムを機械語に翻訳してから実行するものであり，普通の言語はこの方法を用いています（インタープリタやコンパイラというシステム自体は機械語で書かれています）．

このように，どんなプログラムでも，CPUが直接理解できる機械語に翻訳して実行されます．したがって，直接に機械語でプログラムをつくることができれば，それが最も効率的にマイクロ・コンピュータを使用する方法にほかなりません．

しかし，機械語自体は単なる数字のら列にすぎず，人間にとっては大変わかりにくいものです．たとえば，ALレジスタに10Hという数値を代入する命令を機械語で書くと，

```
B0  10
```

となります．これでは，機械には理解できても，人間にはまったく理解するこ

とができません．そこで，このような機械語の命令に1対1で対応し，かつ人間にとっても理解しやすい言語として，**アセンブリ言語**というものがつくられました．

この言語を使用すると，先の命令は，

MOV　AL,10H

と書かれます．はじめの「MOV」が数値などを移動（move）するという命令の種類を表わし，後の「AL，10H」が10HをALに代入するという具体的な動作を表わしています．このような表記法であれば，人間にもわかりますね．

アセンブリ言語
ニーモニック
オペランド

このように，機械語の命令の内容を人間に理解しやすい形で表わした言語をアセンブリ言語と呼んでいます．

アセンブリ言語の命令は，「どうする」という命令の種類を表わした部分（これを**ニーモニック**と呼ぶ）と，「どこからどこに」とか「何をどこに」といった命令の動作する場所を表わした部分（これを**オペランド**と呼ぶ）とに分けられます．

アセンブリ言語の命令の名前は，英語を基本につくられていて，たとえば，

MOV　：move　データを動かす
ADD　：addition　データを加える
JMP　：jump　指定したアドレスにジャンプする

といった形になっていて，命令を覚えるのが容易になっています．

このアセンブリ言語と機械語とは1対1に対応しているので，アセンブリ言語で書かれたプログラムは，すぐに機械語に直すことができるし，逆に機械語の命令をアセンブリ言語に直すこともできます．したがって，アセンブリ言語を学ぶことが，とりもなおさず機械語を勉強することになるわけです．

このようにアセンブリ言語で書かれたプログラムを機械語に直すことを「アセンブルする」といい，そのためのシステムをアセンブラと呼びます．

PC-9801は，8086のためのアセンブラ，ディスアセンブラを内蔵しているし，CP/M-86やMS-DOSに付属しているアセンブラ，ディスアセンブラも使用することができます．これらのアセンブラやディスアセンブラは，細かいところでは違いがありますが，ほとんどが共通していて，8086を開発したメーカであるインテル社のアセンブラ表記法に従っています．

本書では，それらの中で最も身近であり，初心者でもすぐに使用することができるPC-9801内蔵のアセンブラ，ディスアセンブラを用いて，アセンブリ言語の学習をしていきます．また，アセンブリ言語の学習が即機械語の学習になるわけですから，本書では特にアセンブリ言語と機械語という言葉の使い分けはしていません．

なお，アセンブラ，ディスアセンブラの機能としては，内蔵のものよりCP/

M-86やMS-DOSに付属したものの方が優れていますから，本書で一通り学習した後，これらのOS上のアセンブラに移行されることをおすすめします(ただし，これらのアセンブラは，文法的に若干の学習が必要です)．

2 機械は命令をどのように理解するか

さて，アセンブリ言語を使用すれば，機械語のプログラムをつくることができることがわかりました．そこで次に，最も簡単な1＋1を求めるプログラムを例にとり，機械が命令をどのように理解していくのかということについて考えていきます．

まずこのプログラムを，ALレジスタに1を代入しておき，さらにそれに1を加えるという形で作成します．

これをアセンブリ言語で書くと，

```
MOV   AL,1    ←――  ALレジスタに1を代入
ADD   AL,1    ←――  ALレジスタに1を加える
```

となります．

アセンブラを用いてこれを機械語に翻訳すると，

```
B0  01  04  01
```

となります（アセンブラの使い方については次節の「**モニタの使い方**」のところで説明します)．

さて，このような機械語の命令がメモリ上のどこかのアドレスから順に書かれていたとします．メモリは1バイト（8ビット）単位ですから，全部で4バイトになります．

8086は16ビットCPUですが，命令は8ビットを基本に構成されています．そこでCPUは，指定されたアドレスから1バイトずつ命令を解読していきます．

この場合であれば，CPUはまずB0というデータを取り出し，これを命令として解釈します．すると，これはALレジスタに対するMOV命令であるということがわかります．

それと同時に，次の1バイトが命令ではなく，ALレジスタに代入する数値であることがわかるので，次の1バイト（01）を読み込みます．そして，命令に従ってALレジスタに01という数値を代入します．

それが終わると，CPUは次の1バイトを読み込み，再びこれを命令として解釈します．04という命令は，ALレジスタに対するADD命令であり，次の1バイトが加える数値であることがわかるので，次の1バイトを読み込み，ALレジスタに加えます．すると，この時点でALレジスタの内容は2になります．

CPUは，このようにして順次命令を解釈して実行していきます．

3 命令の種類と命令長

ところで，8086の命令はいつも上記のように命令と数値から成っているのかというと，そうとは限りません．たとえば，

```
88  D8
```

は，

```
MOV    AL, BL
```

という命令であり，BLレジスタの内容をALレジスタに転送（代入）する命令です．はじめの「88」がALレジスタに他のレジスタの内容を転送する命令であることを表わし，2バイト目の「D8」がBLレジスタから転送することを示しています．

また，命令の長さ（バイト数）も命令によって異なります．たとえば，

命令の長さは可変長

```
MOV    AX, 1
```

という命令は，

```
B8  01  00
```

という3バイトの命令であり，

```
PUSH  AX
```

という命令は，

```
50
```

という1バイトの命令になります．

長い命令としては，

```
MOV    WORD  [1234], 5678
```

というような命令があり，

```
C7  06  34  12  78  56
```

という6バイトの命令となります．

CPUは，指定されたアドレスから命令を解読，実行していきますが，その際，このように命令の種類によって変化するバイト数に合わせて次に実行する命令の格納されているアドレスを求めていきます．

はじめのような4バイトのデータの並びが，2バイトずつの命令の連続として解釈されたわけです．

したがって，きちんと予定通りのアドレスから始めた場合にはよいのですが，もしはじめのアドレス指定を間違えると，まったく別の命令として解釈されてしまうことになるので，注意が必要です．

4 ハンド・アセンブル

このように，アセンブリ言語で書かれたプログラムを，機械語の命令に直す作業は，通常，アセンブラというプログラムが自動的に行なってくれるので，アセンブリ言語でプログラムを作成してしまえば，あとはそれがどのような機械語に変換されるのかは，ほとんど気にする必要はありません．

しかし，何かの理由でアセンブリ言語を人間が機械語に翻訳する必要のある場合もあります．そのような場合には，アセンブリ言語の命令一覧表から対応する機械語コードを見つけ出し，さらにデータやアドレスなどの計算を人手で行なわなければなりません．このような形でアセンブリ言語を機械語に翻訳する作業を，ハンド・アセンブルと呼びます．

ハンド・アセンブル

Z80や6809などの8ビットCPUでは，ちょっとしたプログラムをつくるときなど，よくハンド・アセンブルしている人をみかけましたが，8086のような16ビットCPUでは命令も複雑になり，ハンド・アセンブルすることは極めて困難なので，よほどの事情がないかぎり，行なわれていません．

また，その逆に人手で機械語をアセンブリ言語に直すこともありますが，これもほとんど必要ないといえるでしょう．

ただ，そのような必要が生じた場合のことも考えて，本書では巻末にアセンブル，ディスアセンブルの一覧表を掲載しておきました．この表を使用すれば従来の参考書の一覧表より容易にハンド・アセンブル，ディスアセンブルができるはずです．PC-9801のような便利なアセンブラ，ディスアセンブラを内蔵していない機種で8086の勉強をされようとする方にとっても，有用ではないかと思います．

3 モニタの使い方

前節で，機械語のプログラムを作成するためにはアセンブラを使う必要があるということ，そしてPC-9801はモニタ上にアセンブラを持っていて，これを用いれば簡単な機械語プログラムを組むことができるということを知りました．

そこで，ここではモニタとは何か，またその使い方について，述べることにします．

1 モニタとアセンブラ

モニタとは，一言でいえば，直接メモリ上のデータを読み書きするためのプログラムのことです．最近のパーソナルコンピュータでは，一般にBASICがROMとして標準装備されていますが，それと併せて直接メモリを操作するためのモニタが標準装備されています．

初心者には内蔵モニタが最適

このモニタを使用することにより，BASICでは実現することのできなかった高度なパソコンの操作を行なうことができます．特にPC-9801のモニタは，他の機種と比べてはるかに高機能であり，ディスクBASICを使用することによって，アセンブラ，ディスアセンブラなども使用できます．つまり，従来はCP/M-86に付属しているASM86などを使用しなければできなかった機械語のプログラムが，PC-9801の本体とディスクBASICさえあればできるようになったわけであり，機械語がよりいっそう私達にとって身近になったということです．

本書において，これから先，機械語を学んでいくにあたっては，このモニタ上のアセンブラを使用していくことにします．初心者にとって，機械語を理解するためには，この方法が最もてっとり早い方法でしょう．

いままで，8086の機械語を学ぼうとする人にとって障害となってきたのは，

CP/M-86上のASM86が比較的難しいということです．このアセンブラは，CP/Mについて一通り理解した上で，アセンブラの細かい文法を覚えなければ，まったくプログラムをつくることができません．

もちろん，モニタ上のアセンブラに比べれば，ASM86などの方がはるかに複雑なプログラムを作成するのに適しています．また，モニタ上のアセンブラでは，何百行もあるようなプログラムを作成するのは困難です．

本書でASM86を使うための基礎力を身につけよう

しかし，本書であえてモニタ上のアセンブラを使用するのは，その使い方を容易に理解することができるからです．読者のみなさんには，一通り本書でアセンブラについて理解した上で，ASM86などの高級なアセンブラを学習されることをおすすめします．

2 ディスクを用意しよう

モニタを使用する前に，これからアセンブラを勉強していくための専用ディスクを1枚準備しましょう．せっかく入力したプログラムがほんの少しのミスによって水の泡にならないように，小まめにプログラムをバックアップしておかなければならないということは，よくいわれることですが，マシン語のプログラムにおいては，さらにそのことが重要になります．

BASICと違って，マシン語のプログラムは入力ミスなどによって暴走してしまうとSTOPキーでは，まず止まることはないでしょうから，ほぼ確実にリセットしなければならなくなります．しかも，場合によってはディスクを破壊してしまうこともあるので，専用の練習用ディスクを準備することが必要なのです．

ディスクは，通常のN88 BASICのフォーマットを行なってあるものであれば，何でも結構です．BASICシステム以外のプログラムは必要ありません．ただし，"setinf.n88" プログラムによって，立ち上げ時の処理を次のようにしておくと便利です．

□内がキーボードから入力する部分

```
RUN"setinf.n88"
Set up info sector

Mount a system disk on drive#? 1
Sure(y/n)? Y

How many files(0-15)? 2
Text for init?
CLEAR,&H1B00:DEF SEG=&H1B00

# of files to allocate is  2
```

```
Text for init is :
CLEAR,&H1B00:DEF SEG=&H1B00

Mount a system disk on drive# 1
Type RETURN if ready
```

ここで自動的にディスクがアクセスされ，必要な情報がディスクに書き込まれます．つまり，使用するファイルの数を2個に，CLEAR文によりBASICの使用するメモリの上限を1B000Hに，機械語プログラムを入力するセグメントベースを1B000Hに設定するわけです．

今後は，こうして作成したディスクによってシステムを立ち上げてあるものとして述べていきます．したがって，このディスク以外で立ち上げた場合には，このディスクを入れてリセットをかけるか，または，

```
CLEAR,&H1B00:DEF SEG=&H1B00
```

を入力してから練習を始めるようにしてください．

また，setinf.n88プログラムによって，画面の初期設定やFILESの実行，プリンタの初期設定なども入れておくと便利です．

3 モニタ・コマンドの使い方

それでは，いよいよモニタの使い方の説明に入ります．まず，先に述べた方法で作成したディスクを使ってシステムを立ち上げてください．

なお，ここでの練習は連続して行なうことを前提としているので，途中から再開したり，リセットをかけてしまった後は，はじめからやり直してください．

まずはじめに，モニタに入るために，

```
Ok
MON
h]
  └カーソル
```

と入力します．「h］」という表示の次にカーソルが点滅しているはずです．このh］という表示が，モニタモードであるということを示しています．h］の"h"は，hexadecimal，すなわち16進数の入力モードであることを表わしています．つまり，このモードで入力された数は，すべて16進数であるとして解釈されるということです．

モニタモードは16進入力

それでは，以下，主要コマンドの使い方を練習することにしましょう．

Cコマンド(チェンジ・セグメント・ベース)

Cコマンドは，Change segment baseという意味で，これからモニタで使用するセグメント・ベースの変更を行ないます．セグメント・ベースについては前にも説明しましたが，これからモニタが使用する64Kバイトの領域のセグメント・ベース・アドレスを設定するわけです．

まず，ただ単にCだけを入力してみましょう．

```
h]C↵
1B00
h]■
```

1B00という値が表示されたはずです．これは，16進5桁のうちの上位4桁のみが表示されているので，つまり現在のセグメント・ベースが1B000Hであることを示しています．前に指定したディスクで立ち上げていればDEF SEG文によってセグメント・ベースが1B000Hになっているはずですから，これは当然のことといえます．

次に，セグメント・ベースを0Hに設定してみましょう．

```
h]C0↵
h]■
```

これで，セグメント・ベースが0Hに設定されました．念のために調べてみましょう．

```
h]C↵
0000
h]■
```

ちゃんと，0Hがセグメント・ベースになっていますね．

なお，モニタモードで使用される数値は，すべて16進数であると決まっているので，モニタのリストでは16進数の後に付ける"H"は付けません．入力のときにも付けてはなりません．本書では，解説の中では16進数にはHを付け，リスト中ではHを付けないことにしています．

Dコマンド(ダンプ・メモリ)

次は，メモリの内容を一度に見るためのDコマンド（Dump memory）について説明します．

Dコマンドでメモリをみるためには，次のようにはじめのアドレスと終わりのアドレスを指定します．

```
hJD0,FF
0000 2B 08 80 FD 2B 08 80 FD E5 07 80 FD 2B 08 80 FD
0010 2B 08 80 FD C5 12 20 F3 46 1D 20 F3 2B 08 80 FD
0020 5D 19 80 FD 74 0D 80 FD 2C 08 80 FD 2C 08 80 FD
0030 E9 16 80 FD 2C 08 80 FD 12 26 80 FD 2C 08 80 FD
0040 2C 08 80 FD 2C 08 80 FD 54 05 00 D6 58 03 00 D7
0050 2C 08 80 FD 2C 08 80 FD 2C 08 80 FD 2C 08 80 FD
0060 B2 09 80 FD B8 14 80 FD 04 23 80 FD 84 19 80 FD
0070 04 18 80 FD 2B 08 80 FD 0A 00 00 E8 00 2A 00 D8
0080 64 11 00 E8 64 11 00 E8 64 11 00 E8 64 11 00 E8
0090 64 11 00 E8 64 11 00 E8 64 11 00 E8 64 11 00 E8
00A0 64 11 00 E8 64 11 00 E8 64 11 00 E8 64 11 00 E8
00B0 64 11 00 E8 64 11 00 E8 64 11 00 E8 64 11 00 E8
00C0 64 11 00 E8 64 11 00 E8 64 11 00 E8 64 11 00 E8
00D0 64 11 00 E8 64 11 00 E8 64 11 00 E8 64 11 00 E8
00E0 64 11 00 E8 64 11 00 E8 64 11 00 E8 64 11 00 E8
00F0 64 11 00 E8 64 11 00 E8 64 11 00 E8 64 11 00 E8
```

はじめのアドレスだけを指定すると，そこから16バイトだけ表示されます．

```
hJD0
0000 2B 08 80 FD 2B 08 80 FD E5 07 80 FD 2B 08 80 FD
hJ
```

簡単なので，普通はこの方法がよく使われます．行の途中からの指定もできます．

```
hJD5
0005 08 80 FD E5 07 80 FD 2B 08 80 FD 2B 08 80 FD C5
hJ
```

長い領域を指定すると，画面上に次々とスクロールしながら，表示されていきます．

表示を一時止めるには，CTRL-S（CTRLキーとSキーを同時に押す）を使います．

また，表示を中断するには，CTRL-C（CTRLキーとCキーを同時に押す）またはSTOPキーを使用します（ESCキーはきかない）．

一時表示を止めたときには，中断するキー以外のキーを押すと表示が再開します（COPYキーを押すと画面がプリンタにコピーされる．この場合はSTOPキーなどで表示が再開する）．

Lコマンド(ディス・アセンブル)

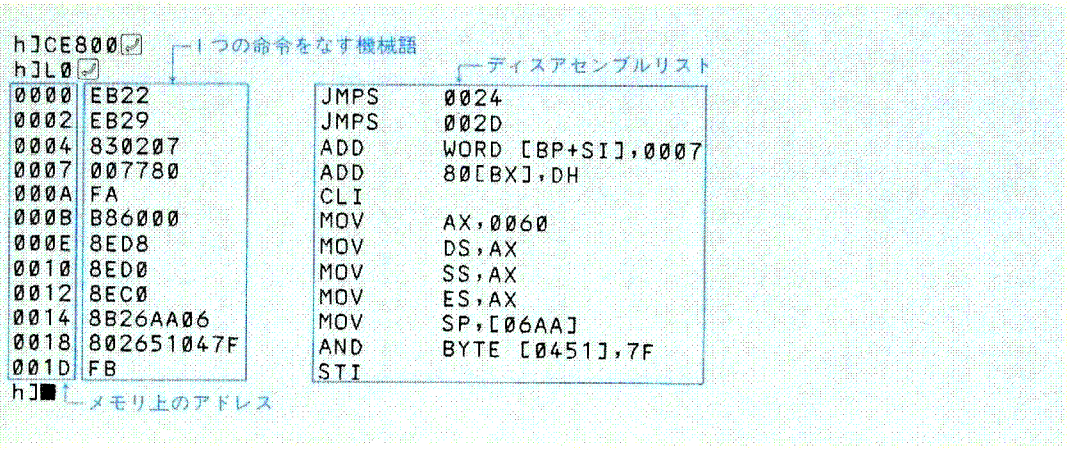
Lコマンドは，メモリの内容をディス・アセンブルするコマンドです．

このコマンドを用いると，メモリ上に格納されたプログラムがどのようなプログラムであるのかを解読することができます．

それでは，BASIC・ROMの格納されているセグメントである0E8000Hにセグメントベースを設定し直した上で，先頭から少しディス・アセンブルしてみましょう．ディス・アセンブルする開始アドレスをLの次に入力します．ここで

BASIC・ROMは0E8000Hから

は０を入力します．



このように，指定したアドレスから12行分だけディス・アセンブルして，画面に表示されます．一番左側が機械語のはじめの命令の格納されているアドレス，その隣りがひとつの命令を構成する機械語の列，そして右側がその機械語のディス・アセンブルリストです．

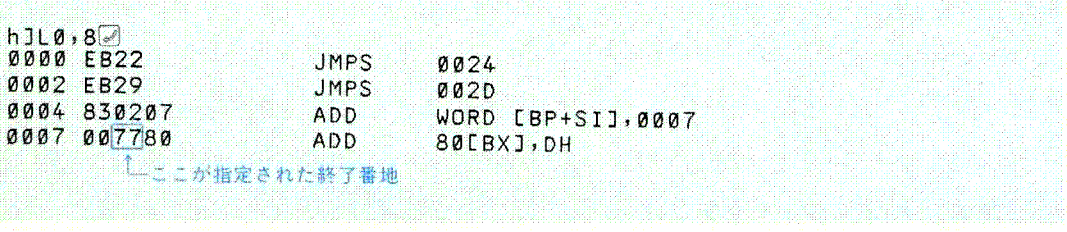
前にも説明したように，機械語は命令の種類によってバイト数が異なります．

なお，ディス・アセンブルリストの読み方については，後で学習していくことにします．

一度Ｌコマンドで表示を行なった後，すぐに続けてリターンキーのみを押すと，続きの12行分のディス・アセンブルリストが表示されます（この機能は，ＬコマンドとＤコマンドのみにある）．また，開始と終了のアドレスを両方指定することもできます．この場合，終了アドレスを含んだ命令まで逆アセンブルして止まります．

リターンキーのみで連続して表示

たとえば，次のようになります．



▶入力の修正法◀

さて，ここまでみなさんは正しく入力することができましたか．もう間違えてしまった方もいるかもしれません．間違えた方は，いつものように入力ミスを直そうとして[BS]キーを押したとき，はてな，と思われたでしょう．そうです．実は，モニタモードでは[BS]キーは使えないのです．それではどうやって修正すればよいのでしょうか．

BSキーでは修正できない

入力はカーソル位置まで

モニタモードでの修正には，カーソルキーを用います．それも，←と→の2つだけです．入力を間違えたら←キーでカーソルをその文字の上まで移動し，入力し直します．修正ができたら→キーで行の最後までもっていき，リターンキーを押します．その際に注意しなければならない点として，次の2つがあります．

① **モニタモードでは，行のはじめから現在カーソルのある位置のすぐ左の文字までを命令として解釈する．したがって修正後，カーソルは正しい位置まで持ってこなければならない．**

たとえば，BASICでは，

```
LUST■
```

と間違えて入力したときに，"U"の位置までカーソルを持っていき，"I"と入力した後，

```
LIST
```

というようになっているときにリターンキーを押しても，ちゃんとそのカーソルを含む行全体を解釈してくれました．しかし，モニタモードでは，たとえば，

```
hJL0,10■
```

となっている"1"を"2"に換えようとして，

```
hJL0,20↵
```

としたとします．するとこれは，カーソルのすぐ左の文字までが入力されたものと解釈されて，

```
hJL0,2↵
```

としたことと同じになります．試してみましょう．

```
hJL0,20↵
0000 EB22          JMPS    0024
0002 EB29          JMPS    002D
```

この場合，修正した後，→キーを用いて0の後までカーソルを持ってきた上でリターンキーを押せば正しく入力されます．なお，このことを逆用して文字を入れすぎた場合の修正の手間を省くこともできます．

② **モニタモードでは，余分な空白を入力してはならない．**

たとえば，BASICでは，

```
LIST␣■
```

としてリターンしても，ちゃんと命令として解釈してくれましたが，モニタモードでは，

```
h]L0,20␣■
```

としてリターンすると，

```
h]L0,20␣↵
?
h]■
```

と表示され，正しく解釈されません．

以上の①，②は，モニタモード中ではすべて共通した事柄です（ただしAコマンド（アセンブラ）を使用しているときは，空白の数に柔軟性のある場合がある）．

余分な空白を入れてはならない

Aコマンド(アセンブラ)

Aコマンドは，Assemblerという意味を持ち，アセンブリ言語を用いてメモリ上に機械語のプログラムをつくっていくためのコマンドです．本書では最も活躍するコマンドといえるでしょう．

Aコマンドは，アセンブルした機械語を格納する最初の番地を指定すると，次々と入力を受け付け，アセンブルモードを終了するまで続けて何行でもアセンブルすることができます．

実際に試してみましょう．機械語コードを格納するセグメントは1B000Hとし，最初からアセンブルを開始します．

```
h]C1B00↵
h]A0↵
0000                    ■
```

さて，どうなりましたか．行の始めに0000とアドレスが表示され，行の途中でカーソルが点滅しているはずです．そこで，キーボードから次のように正しく入力してください(␣はスペースの意味)．

```
0000                    MOV␣AL,2■
```

入力を間違えたときには，前述の方法で訂正してください．ここでリターンキーを押すと，

```
0000 B002            MOV AL,2↵
0002                 ■
```

アドレスの右に何やら数字が現われましたね．これが右のアセンブリ言語を機械語に翻訳したものです．ここのスペースは，翻訳された機械語を表示するために，意識的にあけられていたのです．正しく翻訳することができると，次の機械語の格納されるアドレスを表示して，カーソルが点滅しています．そこで，次の行を入力します．

```
0000 B002            MOV AL,2
0002 2E              CS:↵
0003                 ■
```

入力できましたね．このようにして，次々と入力していくことができます．

ここで，間違った入力をしてみましょう．たとえば，アセンブラにない命令を入力してみます．

```
0003                 A↵
?
0003                 ■
```

ブザーが鳴って，もう一度同じアドレスが表示されましたね．このように，入力を間違えてアセンブルできなかったときには，もう一度同じ行の入力を要求してくるので，入れ直すことができます．そこで，どんどん入力していきましょう．次のように入力してください．

```
0000 B002            MOV AL,2
0002 2E              CS:
0003 8A1E0001        MOV BL,[100]
0007 00D8            ADD AL,BL
0009 2E              CS:
000A A20101          MOV [101],AL
000D F4              HLT
000E                 ■
```

さて，ここまで入力して終わりにしたいのですがどうすればよいでしょうか．終わりにするときには，何も入力せずにリターンキーだけを押します．すると，

```
000D F4              HLT
000E                 ↵
h]■
```

となって，普通のモニタモードにもどります．

それでは本当に正しく入力されているかどうかを，Lコマンドで確認してみましょう．

```
hJL0
0000 B002        MOV     AL,02
0002 2E          CS:
0003 8A1E0001    MOV     BL,[0100]
0007 00D8        ADD     AL,BL
0009 2E          CS:
000A A20101      MOV     [0101],AL
000D F4          HLT
000E 0000        ADD     [BX+SI],AL
0010 0000        ADD     [BX+SI],AL
0012 0000        ADD     [BX+SI],AL
0014 0000        ADD     [BX+SI],AL
0016 0000        ADD     [BX+SI],AL
hJ
```

間隔が調整されている

桁数を合わせるために0が補われている

命令が格納されていない部分は00で埋まっているが，ディスアセンブラは，これも命令だと解釈して翻訳してしまう．実際はこの命令は実行しないので関係ない．

どうですか．間隔が調整され，ところどころに0が補われていますね．Lコマンドでディス・アセンブルすると，MOVのようなニーモニック（命令の機能を表わす）の部分と，AL,02のようなオペランド（命令の作用するレジスタやアドレスなどを示す）との間に，適当な間隔があけられます．Aコマンドによって入力するときには，この部分の空白はひとつ以上あれば，いくつでもかまいません．

1行目の02のように，数字の前に0が補われるのは桁数を合わせるためであり，ALレジスタは1バイト長なので，16進数2桁にするために0が補われます．また，3行目の［0100］は，データの格納されているアドレスを表わしています．アドレスは2バイトで表わされるので，16進4桁にするために0が補われています．入力のときには，前に行なったように，先頭の0は省くことができます．

上位桁の0は省略可能

正しく入力できたことを確認したら，次に進んでください．また間違いに気付いた人は，入れ直してください．たとえば，0007H番地を間違えていたとしたら，

修正の方法

```
hJA7
```

として入力し直してください．生成した機械語のバイト数が同じで，前と同じく次の行が0009H番地であれば，そこでリターンキーを押して修正完了です．もしこれがずれていたら，ここから後を全部入れ直さなければなりません（誤りは早期発見が重要！）．

Gコマンド（実行）

Gコマンドは，Go，つまり「実行せよ」という命令です．この命令を使うと，

メモリ上にあるプログラムを実行することができます．

先に作成したプログラムは，「ALレジスタに2を代入し，BLレジスタに100H番地の内容を転送し，この2つを加えた結果を101H番地に格納する」というものです(使用されるアドレスはコードセグメント内にあるものとします)．

Gコマンドは，次のように使用します．

G開始番地，終了番地

Gコマンドでは終了番地を忘れないように

モニタで作成したマシン語プログラムを実行する場合には，このように開始番地と終了番地を両方とも指定する必要のある場合がほとんどです（終了番地は2つまで指定できる）．終了番地を指定しないと，実行が終了してもモニタモードにもどってこれなくなることがあります．このような場合には，リセットをかけなければならず，せっかく入力したプログラムが消えてしまうので，注意が必要です．

本書のプログラムでは，その都度実行の仕方が書かれているので，それに従ってください．普通はプログラムの最後に"HLT" 命令（HaLT：停止せよ）を書き，このアドレスを指定します．後で説明しますが，実はこのHLT命令を本当に実行してしまうとCPUの動作が止まってしまい，ウンともスンともいわなくなってしまうので，必ずこのHLT命令のあるアドレスを，終了番地（ブレーク・ポイントということもある）として指定することが必要です．

それでは，この命令を使用してみましょう．

```
hJG0,D
*1B00:000D ←――― 実際に実行の終了した番地
hJ■
```

実行の終了した番地が表示されて，モニタモードにもどります．ここで表示されるアドレスは＊に続いて，左側がセグメント・ベース，右側がオフセット・アドレスです(あまり気にしなくてもかまいません)．

一瞬のうちに終わってしまったので，本当に実行したのかどうか不安ですね．Dコマンドを使ってメモリの内容を読み出して確認してみましょう．

```
hJD100
0100  00 02 00 00 00 00 00 00  00 00 00 00 00 00 00 00
hJ■
```

どうですか．100H番地の内容が00Hなので，これに2を加えた値である02Hが101H番地に格納されています．

Sコマンド(セット・メモリ)

Sコマンドは，直接にメモリの内容を書き換えるコマンドです．

このコマンドを使用してメモリの内容を書き換えてみましょう．まず，Ｄコマンドで，200Ｈ番地以降のメモリに何も入力されていないことを確認してください．

```
hJD200↵
0200 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
hJ■
```

次にＳコマンドを使用して，この部分を書き直します．１バイト入力すると次のようになります．

```
hJS200        ←この２桁を入力
0200 00-11■
```

一番左側にアドレスが，次に元のメモリの内容が表示され，"－"の次でカーソルが待っているので11を入力します．すると，上のように入力した数値の次でカーソルが点滅しているはずです．ここで**スペースバー**を押すと，

```
0200 00-11␣00-■
```

と表示され，次のバイト(0201Ｈ番地)の書き換えが可能になります．連続して次のように入力してください．

```
0200 00-11␣00-22␣00-33␣00-44■
```

ここでリターンキーを押すと，モニタモードにもどります．Ｓコマンドで，上記のように入力をしている最中は，16進数字（０～Ｆ）とスペースバー，リターンキー以外のキーは受け付けなくなります．カーソルキーもきかないので，修正はできません．

Ｓコマンドは修正できない

なお，修正の必要のない番地にきたとき，スペースバーのみを押すと，次々とスキップするので，途中をとばして先の番地の修正もできます．

さて，それでは入力された内容を確かめてみましょう．

```
hJD200↵
0200 11 22 33 44 00 00 00 00 00 00 00 00 00 00 00 00        "3#
```

次は，このコマンドを用いて，先のプログラムに数値を代入し計算させてみてください．

```
hJS100↵
0100 00-AB↵
hJG0,D↵
*1B00:000D
hJ■
```

数値0ABH（16進数を表記する際に，最上位の桁がアルファベットのときには，さらにその上に0を付けて表記する）に，2をたしたわけですから0ADHになっていればいいはずですね．のぞいてみましょう．

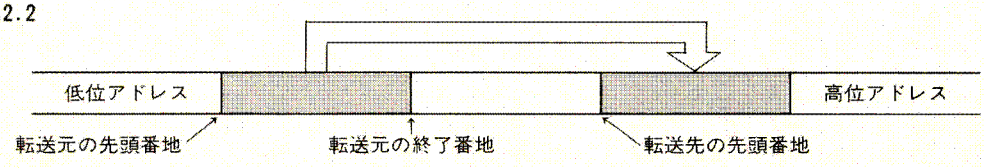
```
h]D100
0100 AB AD 00 00 00 00 00 00 00 00 00 00 00 00 00 00        *
h]■
```

ちゃんと実行されていましたね(AB＋2→AD)．

Mコマンド(ムーブ・メモリ：ブロック転送)

このコマンドは，メモリ上のデータをひとまとめにして，別のアドレスに転送するコマンドです．図2.2のように使用してください．

図2.2



※転送方向は高位アドレス側，低位アドレス側へのどちら向きも可能．転送する領域が重なることも可能である．

使用法は，**M転送するメモリ領域の先頭アドレス，転送するメモリ領域の最終アドレス，転送先の先頭アドレス**です．

ここで，前に作成したプログラムをブロック転送してみましょう．

```
h]M0,D,1000↵
h]
```

さて，どこに転送されたでしょうか，1000H番地からですね．確かめるためには，Lコマンドを使います．

```
h]L1000,100D↵
1000 B002          MOV     AL,02
1002 2E            CS:
1003 8A1E0001      MOV     BL,[0100]
1007 00D8          ADD     AL,BL
1009 2E            CS:
100A A20101        MOV     [0101],AL
100D F4            HLT
h]■
```

もちろん，元のプログラムはそのまま残っています．このようにして，プログラムやデータを転送することができます．プログラムの場合には，転送したものが必ずしも動作するとは限りませんが，データの転送や，プログラムの一時退避(後述)には有効です．

転送領域が重なってもよい

元のプログラムは残っている

Ｅコマンド(エディト・メモリ：編集)

Ｅコマンドはメモリの内容を書き換えるコマンドですが，Ｓコマンドが１バイトずつメモリを書き換える命令であったのに対して，Ｅコマンドでは１画面分(256バイト)のメモリを，画面上で自由に書き換えることができます．

Ｅコマンドを使用するときは，はじめのアドレスを指定します．

```
hJE100↵
```

すると，いままで表示されていた画面がクリアされ，一面に256 バイト分のメモリの内容が表示され，最初のデータ上でカーソルが点滅しています．

```
0100 AB AD 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ｵﾕ
0110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

ここで，たとえば"11"と入力すると，画面上のデータが入れ換わり，カーソルは次のバイトの頭で点滅しています．

```
0100 11 AD 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ﾕ
```

そこで，次に"22"と入力すると，同様に画面上に22が現われ，カーソルが移動します．次々と数値を入力し，右端までくると次の行に移ります．

```
0100 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF 00   "3DUfwﾘｪｻﾌﾝ/
0110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

修正を行なう場合には，カーソルキーが使えます．カーソルキーで目的のデータの位置までカーソルを移動させ，直接数値を入力するだけで完了です．カーソルキーは，←，→だけではなく，↑，↓も使用できます．ちょっとやってみましょう．いま入力した最初の"11"を"12"に変更します．

↑キーを押すと，カーソルが上の行に移動します．次に→キーで同じバイトの下位の桁にカーソルを動かします．ここで"２"を押せば修正できます．

```
0100 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF 00          "3DUfwﾌｪｻﾌﾝ/
```

このようにして，どんどん入力したり修正したりすることができます．本に書かれているダンプリストを入力するときなどに便利です．

エディタでの編集は自由自在

さらに，このモードでは[Roll Up]，[Roll Down]キーを使用して，画面上に表示されている部分の次のメモリや，前のメモリを修正することができます．

入力が終了したら，カーソルの位置に関係なく，[ESC]または[STOP]キーを押すと，編集画面が消えて元のモニタモードにもどります．いままで入力したものは，すべて入力されているはずです．

なお，編集中にキーの使い方がわからなくなったときは，[HELP]キーを押すと，一覧表が表示されます．

Fコマンド(フイル・メモリ)

このコマンドは，メモリ上の連続した領域を同じデータで埋める命令です．使用法は，次のようになります．

F開始アドレス，終了アドレス，埋めるデータ

実行してみましょう．前のコマンドの実習の結果，100H番地からいろいろな値が入っています．

そこで，この1行をすべて"7F"にしてみましょう．

```
hJD100
0100 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF 00          "3DUfwﾌｪｻﾌﾝ/
hJF100,10F,7F
hJD100
0100 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F
```

1行分が，7FHで埋められています．

メモリをクリアする方法

この命令は，メモリ上の一定部分を00で埋めてクリアするときに，よく使用されます．本書にもたくさんの例題や練習問題がありますが，それらのプログラムをキーボードから打ち込む前に，この命令を用いてメモリをクリアしておきます．

使い方は，次のようになります（ただし，いまは打ち込まないでください．プログラムが消えてしまって，次の解説が行なえなくなってしまいます）．

```
hJF0,1FF,0 ↵  (メモリの0番地から1FFH番地までがクリアされる)
```

Xコマンド(イグザミン・レジスタ)

Xコマンドは，8086の全レジスタの内容を表示・変更するコマンドです．命令を実行した後，各レジスタにどのような値が格納されているのかを調べるの

に使います．プログラムの動作がどうもおかしいというときなどに，プログラムを途中まで実行させてから，このコマンドでレジスタの内容を調べることにより，思ったとおりに動作しているかどうかを調べることができます．

実際に，プログラムを実行した後の様子をみてみましょう．

```
hJG0,D
*1B00:000D
hJD100
0100 7F 81 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F
hJ■
```

100H番地の内容である7FHに2Hが加えられ，結果の81Hが101H番地に格納されていますね．このときのレジスタの内容は，

```
hJX↵
FL O---S-AP- ←――実行の結果，いくつかのフラグがセットされている
AX 0081 ←――加算結果の81HがAXレジスタの下位8ビットであるALレジスタに残っている
BX 007F ←――加算する数値の7FHがBXレジスタの下位8ビットであるBLレジスタに残っている
CX 0000
DX 0000
SP F7F6
BP 0000
SI 0000
DI 0000
CS 1B00 ←――プログラムの書かれているセグメントベースが1B00Hであることを示している
DS 0000
SS 0060
ES 0000
IP 000D ←――実行終了番地である000DHが入っている
hJ■
```

どうです．レジスタにはいろいろな値が残っていますね．FL（フラグ・レジスタ）だけは，ひとつひとつのビットに意味があるので，1となっているビットの名称の頭文字を表示します．0となっているビットは，"－"表示されます．各ビットの意味は，**4章**の「**フラグ・レジスタの働き**」のところで説明します．

FL(FLag register)は，1ビットずつ表示される

このXコマンドによって，レジスタの内容を書き換えることもできます．たとえば，AXレジスタの内容を書き換えるときには，

```
hJXAX↵
AX 0081 ■
```

となり，現在の内容を表示してカーソルが点滅しているので，

```
AX 0081 FFFF■
```

と代入します．ここで，リターンキーを押すと，モニタモードにもどります．

レジスタの内容の書き換え方

しかし，Sコマンドと同様にスペースバーを押すと，次のレジスタの内容が表示されるので，続けて書き換えが可能です．書き換えたくないレジスタは，スペースバーのみを押すとスキップされます．リターンキーのみでモニタにもどります．

```
AX 0081 FFFF ␣←―スペースバーが押されている
BX 007F ␣←―――――　　　〃
CX 0000 ␣←―――――　　　〃
DX 0000 ↵
h]■
```

なお，CSレジスタは書き換えない方が無難です．

それから，フラグ・レジスタの書き換えは，個別に指定します．たとえば，いまOフラグ（オーバーフロー・フラグ）が1になっていますが，これを0にしてみます．

```
h]XO↵
O 1 0
  │ └――― キーから入力した内容
  └―――― 元の内容

h]X↵
----S-AP- ┐
AX FFFF   ┘ 書き換わっている
BX 007F
CX 0000
DX 0000
```

モニタには他にもたくさんのコマンドがありますが，アセンブラでプログラムを組むのに必要なコマンドは，ほぼこれで全部です．その他のコマンド（おそらく必要ないと思いますが）については，マニュアルをみてください．

この後，いくつかのコマンドについて説明し，終わりたいと思います．

Pコマンド(プリンタ・スイッチ)

トグル・プリンタ・スイッチ

このコマンドは，プリンタに表示を出すか否かを決めるものです．ただし，プリンタに出力できるのは，Lコマンドによるディスアセンブルリストと，Dコマンドによるダンプリストだけです．

プロンプト

このコマンドは，一度実行するとプリンタONの状態（出力する状態）となり，もう一度実行するとプリンタOFFの状態（出力しない状態）となります．そのため，トグル・プリント・スイッチとも呼ばれています．プリンタがONかOFFかは，行の始めに出力される"h"の次のマークが"]"であればOFFのとき（つまり，いままでの状態），")"であればONのときであるということを表わしています（このように入力待ちのときに表示される記号をプロンプト（入力促進記号）と呼ぶ）．

ちょっと，やってみましょう．プリンタの電源スイッチを入れてから，次の命令を実行してください．

```
h]P↵ ←――プリンタON
h)D100↵
0100 7F 81 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F _
h)P↵ ←――プリンタOFF
h]
```

（この部分が画面と同時にプリンタに出力される．）

もし，プリンタがなかったり，準備ができていないのに，うっかりとこのコマンドを実行してしまうと，一時実行が停止しますが，そのときは，あわてず騒がずSTOPキー，またはCTRL－Cを押してください．数秒であきらめて，モニタモードにもどってきます．

HELPキー

コマンドを忘れたらHELPキー

このコマンドを知らない人はいませんね．コマンドの使い方を忘れたときには，このHELPキーを押せばよいのです．

なお，表示が1画面に収まりきれず，はじめの部分が消えてしまうので，はじめの部分が見たいときには，素早くCTRL－S，またはCTRL－C，あるいはSTOPキーを押してください．

CTRL－Bコマンド

さて，以上でモニタのコマンドの解説を終わりますが，忘れてはならないのがこのCTRL－Bコマンドです．賢明な読者のみなさんは御存知だろうと思いますが，このコマンドは，モニタモードからBASICモードにもどるための唯一のコマンドです．理由付けはともかく，BASICへもどるからCTRL－Bなんだと覚えておけば，間違いはないようです．

それでは，BASICへもどってください．

```
h]^B
Ok
■
```

ごくろうさまでした．これで終わりといいたいところですが，ちょっと待ってください．せっかく作ったプログラムですから，ディスクにセーブしておかなければなりません．PC-9801では，マシン語のプログラムのセーブもBASICモードで行ないます．以下にその方法について説明します．

BSAVE

マシン語のプログラムをディスクにセーブするには，BSAVE命令を使いま

す．BSAVE命令は，メモリ上のデータを直接ディスクにセーブする命令です．したがって，プログラムに限らず，データなどのセーブもできます．

この命令を使用するのに先立って，プログラムがメモリ上のどこにあるのかということをもう一度確認しておく必要があります．その際，現在使用しているセグメント・ベースの確認も忘れないでください．

SAVE前に必ず確認を

いまの場合，前記の方法により作成したディスクで立ち上げているとすれば，セグメントベースは1B00Hになっているはずです(**注1**)．しかし，念のため，宣言しておいた方がよいでしょう．

以下に，操作の手順を示します．

```
DEF SEG=&H1B00↵
Ok
MON↵
h]L0↵
0000 B002            MOV     AL,02
0002 2E              CS:
0003 8A1E0001        MOV     BL,[0100]
0007 00D8            ADD     AL,BL
0009 2E              CS:
000A A20101          MOV     [0101],AL
000D F4              HLT
000E 0000            ADD     [BX+SI],AL
0010 0000            ADD     [BX+SI],AL
0012 0000            ADD     [BX+SI],AL
0014 0000            ADD     [BX+SI],AL
0016 0000            ADD     [BX+SI],AL
h]^B ←――― CTRL Bで BASICにもどる
Ok
```

さて，プログラムは0H～0DH番地までに入っていることがわかりました．そこで，これをセーブします．方法は，次のとおりです．

BSAVE "プログラム名", 開始アドレス, プログラムの長さ

ここで注意が必要なのは，最後の"プログラムの長さ"です．いままでのコマンドでは，この部分は終了アドレスでしたが，こんどはプログラムの長さです．しかも，ここは"長さ"ですから，0H～0DH番地までをセーブするときには0EHとなります．0H番地から始まるプログラムの場合には，終了番地＋1がプログラムの長さです．

BSAVEだけが"長さ"を指定する

DEF SEG文は恒常的

このプログラムをセーブする方法は，次のようになります．

```
BSAVE"EX5",0,&HE↵
```

注1)DEF SEG文で宣言したセグメントベースの値は，モニタ使用中にCコマンドで設定し直すことができるが，BASICにもどると元のDEF SEG文で宣言した値にもどる．したがって，再びモニタに入ったときには，DEF SEG文で宣言したセグメントベースの値が有効となる．Cコマンドは一時的なもの，DEF SEG文は恒常的なものと覚えておく．

長さに余裕を持ってSAVEしよう

間違いがなければ，ディスクが一瞬カチャッといってセーブが終わるはずです．この第2パラメータが"長さ"であること，さらに，BASICなので16進数の値には"&H"を付けなければならないこと，などがこのコマンドの使いにくさにつながっていることは否定できません．また，前記のようにマシン語の格納アドレスを調べるため，いったんモニタモードにもどって，さらにBASICでセーブする，といった手間がかかるので，筆者としてはモニタコマンドにセーブのコマンドを設けてほしかったと思います．

なお，通常は上記のようにプログラムの長さを厳格にしなくても，少し多目にとって256バイト単位でセーブしてしまえば楽になります．すなわち，

```
BSAVE"EX2",0,256
```

とすればよかったわけです．それほど長いプログラムでなければ，これでも決してディスクの無駄使いにはならないはずです．

BLOAD

この節をしめくくるにあたって，マシン語のロードの方法について解説しましょう．FILESコマンドを使用して前にセーブした"EX2"という名のファイルがディスク上にあることを確認したら，リセットをかけて機械を立ち上げ直してください．この状態では，メモリ上には何もプログラムはないはずです．

そこで次のようにすると，前のプログラムがロードされます．

```
DEF SEG=&H1B00
Ok
BLOAD"EX2"
```

```
MON
hJL0,D
0000 B002          MOV     AL,02
0002 2E            CS:
0003 8A1E0001      MOV     BL,[0100]
0007 00D8          ADD     AL,BL
0009 2E            CS:
000A A20101        MOV     [0101],AL
000D F4            HLT
```

となっていれば，正しくロードできたわけです．もし，こうならなかったという人は，その原因をよく考えてください．当然このプログラムも，以前と同様に実行することができます．

なお，BLOAD命令には，セーブしたときと違う番地からロードする機能，および，ロード後ただちに実行する機能がありますが，本書におけるアセンブラの学習には必要ないので，その使用法についての説明は省きました．

第3章

転送命令とアドレス指定

1　データ転送命令

2　アドレッシング・モード

1 データ転送命令

本節では，いよいよひとつひとつの命令の解説に入ります．コンピュータに計算をさせるためには，外部からデータを入力し，CPUに計算をさせ，結果を出力させなければなりません．CPUの作業の基本はメモリからデータを取り出し，演算を行なったあと，再びメモリに格納するといった動作です．

メモリからデータを取り出してCPU内のレジスタに転送したり，レジスタからメモリにデータを転送したりする命令がデータ転送命令です．

データ転送命令では，レジスタ間でのデータ転送，レジスタやメモリへの数値代入などを行なうことができます．データは8ビットまたは16ビットで，使用するレジスタによってどちらが選ばれるかが決まります．長さを指定しなくてはならない場合もあるので，扱っているデータが8ビットなのか16ビットなのか，常に注意しておく必要があります．

1 データ転送命令の記述法

8086のアセンブラでは，データ転送命令は，

```
MOV   DEST , SOURCE
```

という形で記述されます．MOVはデータ転送を意味していて，このように**命令の種類を表わすものをニーモニックといいます。**

ニーモニック
オペランド
ディスティネーション
ソース

またDESTはDESTINATION（目的地）の略でデータの転送先を，SOURCE（出所）はデータの転送元を表わしています．このように，**命令が行なわれる対象となるものをオペランドといいます。**

この命令形式からわかることは，データは右のSOURSEから左のDESTに送られる，ということです．

では，ここで具体例をひとつ示しましょう．

MOV　BX，AX

は，AXレジスタの内容をBXレジスタに転送する，という意味です．これはBASICの代入文の形と同じで，BX＝AXという意味だと考えれば，すぐに慣れると思います．なお，ここでいう「転送」とは，「代入」の意味ですから，転送元の内容はそのまま残り，なくなってしまうわけではありません．

2 メモリ・レジスタの指定法

間接指定法

ここでもうひとつ知っておいてほしいのは，［　］を用いた**間接指定法**です．いくつか実例をあげながら説明していきましょう．

MOV　AX，1234H

とした場合，これは，AXレジスタに1234Hという数値を代入する，という意味になります．ところが，

MOV　AX，[1234H]

とすると，これは，1234H番地のメモリの内容をAXレジスタに代入する，という意味になります．ところで，AXレジスタは16ビットの長さがありますから，データは16ビット長となり，実際には1234H番地と1235H番地の内容がAXレジスタに転送されます．さて，

MOV　AX，BX

とした場合には，AXレジスタにBXレジスタの内容（数値）が転送されるということは，すでに述べたとおりです．それでは，

MOV　AX，[BX]

としたら，どうなるでしょうか．この場合，BXレジスタの値の示すアドレスの内容がAXレジスタに転送されます．もしBXレジスタの値が1000Hであったとすれば，1000H，1001H番地の内容がAXレジスタに転送されるわけです．

3 格納のされ方

命令はコードセグメント

「**アセンブルの使い方**」で述べたように，転送命令は機械語に翻訳されてメモリに格納されていきます．CPUはその機械語をひとつずつ取り出して，命令を実行していくわけですが，この機械語の命令は，必ずコードセグメントで指定されるセグメント内に書かれなければなりません．CPUは，これから実行す

る命令の書かれているアドレスをコードセグメントの値に加えたアドレスから，命令を取り出してきます．これから実行する命令の書かれているアドレスは，命令実行ポインタ（IP：インストラクションポインタ）の値として示されています．たとえば，CS＝1B00H，IP＝1234Hとすれば，CPUは1B000H＋1234H番地の内容を命令として取り出します．もっとも，コードセグメントは一度指定しまうと，変更することはほとんどありませんから，上記のようなアドレスの計算法をいちいち意識する必要はありません．

4 データセグメントの設定

さて，CPUが取り出してきた命令が，次のようなものであったとします．

MOV　AX，[1234H]

このような場合CPUはどこから持ってきたデータをAXレジスタに格納するのでしょうか．実は，データセグメントの1234H番地から持ってくるのです．

データは
データセグメント

このように転送命令では，**データはデータセグメント（DS）から持ってくる**という暗黙の約束があるのです．つまり，命令の格納されているセグメントと，データの格納されるべきセグメントとが違っていることに注意しなければなりません．

たとえばDS＝1C00Hであるとすると，上の命令では1C000H＋1234H＝1D234H番地（と次の番地）からデータを取り出し，AXレジスタに格納します．このようにデータ転送命令では，**メモリの番地が指定されたとき，それはデータセグメントにあたるものとして解釈されます。**

しかし，どうしてもデータやセグメント以外からデータを持ってきたい場合もあります．そのような場合には，次のような命令で行なえます．

CS：MOV　AX，[1234H]

とすると，コードセグメント内の1234H番地のデータをAXレジスタに転送することができます．このような記号をセグメントオーバーライドプリフィックスといい，CS：，DS：，ES：，SS：があります．このような命令は，この記号のすぐあとの命令に限って有効です．この命令はまた，

セグメントオーバーライドプリフィックス

CS：
MOV　AX，[1234H]

のように分けて書くこともできます．

このように8086では，アドレスがどこのセグメントにあるのかに常に注意を払わなければならないのですが，詳しくは「**アドレッシングモード**」の項でもう一度触れることにして，ここで転送命令の整理をしましょう．

使用記号とMOV命令の整理

■使用される記号の整理■

reg：**レジスタ**……AX，BX，CX，DX，AH，BH，CH，DH，AL，BL，CL，DL，SI，DI，SP，BPの各レジスタを指す．特にreg 8 で 8 ビットのレジスタを，reg16で16ビットのレジスタを指すことがある．

sreg：**セグメントレジスタ**……CS，DS，ES，SSの各レジスタを指す．

mem：**メモリ**……8 ビットまたは16ビットのメモリの内容のことで，[○○]の形で表わされる．○○はメモリの存在するアドレスを示す数値，または一部のレジスタ（ポインタレジスタと呼ばれる）である．これについては「**アドレッシングモード**」の項で説明する．

8 ビット，16ビットのメモリはそれぞれ特に，mem8，mem16と表わすこともある．

■MOV命令の整理■

MOV　reg，即値　……レジスタに直接数値を代入する．

MOV　mem，即値　＊……メモリに直接数値を代入する（**注 1**）．

MOV　reg，reg　……レジスタからレジスタに値を転送する．

MOV　mem，reg　……レジスタからメモリに値を転送する．

MOV　reg，mem　……メモリからレジスタに値を転送する．

MOV　sreg，reg　……レジスタからセグメントレジスタに値を転送する（**注 2**）．

MOV　sreg，mem　……メモリからセグメントレジスタに値を転送する（**注 2**）．

MOV　reg，sreg　……セグメントレジスタの値をレジスタに転送する．

MOV　mem，sreg　……セグメントレジスタの値をメモリに転送する．

※この命令群ではフラグは変化しない

注1)この命令の使い方については，「バイトとワード」の項で詳しく説明する．

注2)コードセグメントに対しては，値を転送することは許されない（ただし値を取り出すことはできる）．これは，コードセグメントは現在実行中の命令の位置を示すセグメントなので，これを変更すると次の命令の場所がわからなくなってしまうからである．同様の理由でIPに対する値の転送もできない．CS，IPに値を転送するためには，特別な方法を用いなければならない．

表3.1　可能な転送の組み合わせ

転送方向↙	レジスタ	セグメントレジスタ	メ モ リ	即　　値
レジスタ	○	○	○	○
セグメントレジスタ（CSを除く）	○	×	○	×
メ モ リ	○	○	MOVS命令	○

表3.1からわかることは，

① **レジスタはすべての形の転送および代入が許される．**

② **メモリはメモリどうしの転送以外のすべての形の転送および代入が許される．**

③ **セグメントレジスタはメモリまたはレジスタとの間での転送しか許されない．**

ということです．これは覚えてしまってください（メモリどうしの間での転送は，別の命令〔MOVS命令〕を用いて行なうことができます．これは「**ストリング命令**」のところで説明します）．

例　題

ALレジスタに12H，BLレジスタに34Hを代入したあと，コードセグメントの100H番地にALレジスタの値を，101H番地にBLレジスタの値を転送するプログラムを作れ．

解　答　「**モニタの使い方**」のところで述べた方法で，システムを立ち上げ，モニタモードで以下のプログラムを打ち込みます．

```
h]A0 ↙
0000 B012
0002 B334
0004 2E
0005 A20001
0008 2E
0009 881E0101
000D F4
000E
h]
```

```
MOV AL,12 ↙
MOV BL,34 ↙
CS: ↙
MOV [100],AL ↙
CS: ↙
MOV [101],BL ↙
HLT ↙
↙
```

・実際に入力するのはこの部分だけでよい．
・モニタモードではすべての数は16進数として扱われているので，16進数を表わす"H"は付けていない．

正しく入力されたかどうかを，Lコマンドで確かめてみましょう．

```
h]L0,D↙
0000 B012          MOV     AL,12
0002 B334          MOV     BL,34
0004 2E            CS:
0005 A20001        MOV     [0100],AL
0008 2E            CS:
0009 881E0101      MOV     [0101],BL
000D F4            HLT
h]
```

間隔を調整し，必要な0が補われているのがわかりますね．今後も，特にことわらない限り，Lコマンドで確認したリストの形式でプログラムを書きますが，間隔は最低ひとつあればよく，また上位桁の0は省略してもかまいません．

それでは実行してみましょう．ブレークポイントの設定を忘れないように．

```
h]G0,D↵
*1D00:000D
h]
```

命令が正しく実行されたかどうかを確かめてみましょう．

```
h]D100 ↵
0100 12 34 00 00 00 00 00 00 00 00 00 00 00 00 00 00          4
h]
```

100H番地に12Hが，101H番地に34Hが格納され，正しく命令が実行されたことがわかります．

ここでは8ビットのデータを扱いましたが，次に16ビットのデータを扱ってみましょう．

例題2

AXレジスタに1234Hを代入し，これをコードセグメントの100H，101H番地に転送せよ．また，102H，103H番地に直接数値1234Hを代入せよ．

解　答　今度は16ビットの値を扱うので，一度に2バイト転送する点に注意してください．

```
h]F0,1FF,0    ←――前に入れたプログラムとデータを消すための命令
```

としたあと，次のプログラムを入力してください．

```
                        (キー操作)
0000 B83412             MOV AX,1234
0003 2E                 CS:
0004 A30001             MOV [100],AX
0007 2E                 CS:
0008 C70602013412       MOV WORD [102],1234
000E F4                 HLT
                              ↑――この意味は次の項で説明
000F
h]
```

正しく入力できたら，Lコマンドで確認したあと，実行してみましょう．

```
h]G0,E
*1B00:000E
h]
```

正しく実行されたかどうか，Dコマンドでみてみましょう．

```
hJD100
0100 34 12 34 12 00 00 00 00 00 00 00 00 00 00 00 00        4 4
hJ
```

何だか変ですね．100 H番地に34Hが，101 H番地に12Hが入っています．102H番地と103H番地も逆転していますね．

それではAXレジスタがどうなっているのかみてみましょう．

```
hJXAX
AX 1234↙←何も入れずにリターンキーを押してみる
hJ
```

AXレジスタには正しく1234Hと入っています．いったいどうしたのでしょうか．実はこれが正しく実行された結果なのです．次の項でその理由を説明しましょう．

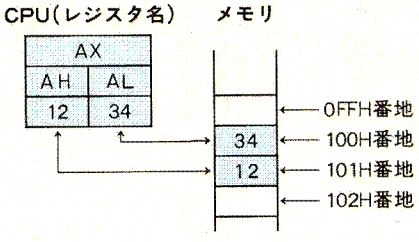
5 メモリ転送と逆ワード

8086の規則では，メモリにワードデータを格納する場合は下位バイトを先に，上位バイトを後に入れることになっています．これが**逆ワード形式**と呼ばれる格納法で，そのため，見かけ上**ワード単位のデータをCPUとメモリ間で転送する場合には，上位バイトと下位バイトが入れ換わるようにみえるのです．**

したがって，**例題2**の場合も規則のとおりに順序が入れ換わって転送されたわけです．

これを図で表わすと，**図3.1**のようになります．

図3.1　逆ワード



この規則は人間にとっては大変わかりにくいのですが，80系のCPUではすべてこの方式をとっています．この方式のことを逆ワード方式といいます．

命令をよくみてみると，0000番地では，MOV AX，1234が，機械語ではB83412と記述されています．これはメモリ上にある命令をCPUに取り出してきたときに，正しく1234とAXレジスタに入れるには，あらかじめ逆ワードにして格納する必要があるからです．

CPUはメモリからAXレジスタに命令を取り出す場合，順番を逆転させてAXレジスタに入れます．さらにAXレジスタからメモリの100H，101H番地に転

送するときに，また順番を逆転させて転送するため，34H，12Hという順番になるわけです．直接数値を代入する場合にも順序が逆転するのは，**CPUが正しく認識できる形のワードデータは，メモリ上では逆転する，**という規則に従っているからです．ちょっとひねくれていますね．ちなみにレジスタ間の転送ではこのようなことは起こりませんし，あとで述べるメモリ間の転送でもこのようなことは起こりません．

逆ワード形式

6 バイトとワード

前のプログラム中で，WORDという単語が現われましたが，これの意味について説明しましょう．

この節の初めに，扱っているデータが8ビット（バイト）か16ビット（ワード）かということは命令に含まれるレジスタで決まる，と述べました．

データがバイトかワードかは使用するレジスタで決まる

したがって，

```
MOV　[1234H],AL
```

とすれば，ALレジスタの内容を1234H番地に転送するバイト転送であり，

```
MOV　[1234H],AX
```

とすれば，AXレジスタの内容を1234H番地と1235H番地に逆ワードにして転送するワード転送であることがわかります．

しかしメモリに直接数値を代入する場合，たとえば，

```
MOV　[1234H],56H
```

としたとき，1234H番地に56Hを転送するのか，1234H番地と1235H番地に，0056Hを（逆ワードにして）転送するのかわかりません．そこでこの場合どちらであるかをはっきりさせるために，

直接メモリにデータを書き込むときにはBYTE指定，WORD指定が必要

```
MOV　BYTE　[1234H],56H
```

または，

```
MOV　WORD　[1234H],56H
```

のように，BYTEまたはWORDを使って，データの長さを指定しなければなりません．試しに，Aコマンドで，

```
MOV　[1234],56　←――入力時にはHを付けない
```

と入力してください．翻訳エラーになったはずです．

このようにレジスタを使わず直接メモリを操作する場合には，必ずデータのタイプとしてBYTEまたは，WORDを指定しなければなりません．あとで述べる算術演算命令や論理演算命令も同様です（ただしどちらか一方のオペランドがレジスタである場合には，この指定を用いるとエラーになります）．

もっとも転送するデータが16進で3桁か4桁の場合には，自動的にワードデ

ータとして扱われますが，将来プログラムにおいて思わぬミスをしないためにも，直接メモリを操作する命令にはBYTEまたはWORDを付ける習慣を身につけておくことが必要です．

練習問題

3—1

データセグメントを1C00Hに設定し，コードセグメントの100H，101H番地に1234Hを代入したあと，その値をデータセグメントの100H，101H番地に転送せよ．

解答

```
0000 B8001C          MOV     AX,1C00
0003 8ED8            MOV     DS,AX
0005 2E              CS:
0006 C70600013412    MOV     WORD [0100],1234
000C 2E              CS:
000D A10001          MOV     AX,[0100]
0010 A30001          MOV     [0100],AX
0013 F4              HLT
h]
```

セグメントレジスタには直接数値を代入することができないので，このようにレジスタを介して行なう．

代入データがワードタイプであることを示す．

CS:の指定は直後の指定のみに有効なので，ここではデータセグメントが用いられる．

解説

実際に実行してみましょう．

```
h]G0,13↵
*1B00:0013
h]D100↵
0100  34 12 00 00 00 00 00 00  00 00 00 00 00 00 00 00        4
h]
```

となり，コードセグメントのオフセットアドレス100H以降に逆ワード形式で，1234Hが入っていますね．さらに，

```
h]C1C00↵←――――セグメントベースを，データセグメントに変更
h]D100↵
0100  34 12 00 00 00 00 00 00  00 00 00 00 00 00 00 00        4
h]
```

となっていますから，データセグメントのオフセットアドレス100H以降にも，同じデータが入っていることがわかります．

なお，新しいプログラムを入れる前に必ず，

h]F0,1FF,0 ↵

を実行し，メモリの必要部分をクリアするのを忘れないようにしてください．

3—2

いちいちCS：プリフィックスを付けなくても，コードセグメント内でデータの転送ができるようにするにはどうしたらよいか，方法を考えよ．

解　答

```
0000 8CC8          MOV AX,CS
0002 8ED8          MOV DS,AX
```

を，プログラムの先頭に入れればよい．

解　説

このプログラムにより，データセグメントとコードセグメントとが一致するので，見かけ上コードセグメント内だけでデータ転送ができるような形になります．前にも説明しましたが，セグメントはオーバーラップしたり，完全に一致したりしてもかまいません．このような方法は，作るプログラムの使用するメモリ空間が狭く，特にセグメントに分ける必要のない場合に使用すると，プログラミングが楽になります．

CS，DS，SS，ESのすべてを一致させることにより，実質上使用するメモリは64Kバイトだけになります．これは8ビットCPUを使用した時のメモリと同じサイズであり，8086上で8ビットCPUのプログラムをシミュレートする場合などに使われます．本書掲載のプログラムでは特にことわりのない限り，上記の方法でデータセグメントをコードセグメントに一致させてからプログラムを実行させます．

セグメントを一致させると便利

3—3

BXレジスタに100Hを代入し，これを用いて100H番地にバイトデータ56Hを代入せよ．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 BB0001        MOV     BX,0100
0007 C60756        MOV     BYTE [BX],56   ←データがバイト単位であることを示し，
000A F4            HLT                      間接アドレスを使用して番地を指定する
```

解　説

間接アドレスの使い方とバイト指定の練習です．バイト指定を行なわないと

アセンブルエラーになります．

では実行してみましょう．

```
h]G0,A
*1B00:000A
h]D100
0100 56 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00          V
```

となり，100H番地にバイトデータ56Hが入っていることがわかります．

3—4

AXレジスタに1234Hを代入し，これを用いて100H番地に12Hを，101H番地に34Hを代入するプログラムを作れ．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 B83412        MOV     AX,1234
0007 88260001      MOV     [0100],AH   ←――指示どおりに転送するためにAXレジスタを
000B A20101        MOV     [0101],AL       AHとALに分けて転送する
000E F4            HLT
```

解　説

AXレジスタはAHレジスタとALレジスタとに分けて使用できます．AXレジスタに1234Hを代入したときのレジスタは図3．2のようになっています．しかし，これをそのままAXレジスタのワードデータとして100H，101H番地に転送したのでは，逆ワード形式のため34H，12H，の順に並んでしまいます．

図3.2

AX	
AH	AL
12	34

そこでAHレジスタとALレジスタの2つに分けて，それぞれを100H番地，101H番地に転送することにより，12H，34Hの順に並べることができます．なお，AH，ALレジスタはバイト単位であることが明確なので，MOVのあとのBYTEは不要です（付けるとエラーとなります）．

では，確認してみましょう．

```
h]G0,E↵
*1B00:000E
h]D100↵
0100 12 34 00 00 00 00 00 00 00 00 00 00 00 00 00 00          4
h]
```

大丈夫ですね．

2 アドレッシング・モード

前章でデータ転送命令を学び，「何を」，「どこに」転送するのかの表わし方にいくつかの方法があることを知りましたね．復習する意味で次に示します．いずれも"AXレジスタ"に，転送（代入）しています．

```
MOV   AX, 1234H      数値1234Hを代入する．
MOV   AX, [1234H]    1234H番地の内容を転送する．
MOV   AX, BX         BXレジスタの内容を転送する．
MOV   AX, [BX]       BXレジスタの値の示すアドレスの内容を転送する．
```

データを持ってくる方法は，この他にもいくつかあります．このようにデータを持ってきたり，または持っていったりする場所を示す方法のことを，アドレッシング・モードといいます．

アドレッシング・モードとはデータを転送する場所を示す方法

1 アドレッシング・モードの種類

アドレッシング・モードは大きく分けて，イミディエイト・モード，直接モード，間接モードの3つに分けられます．

イミディエイト・モードとは，前述の転送命令の最初の例のように，数値そのものを代入する場合です．

直接モードとは，同例3番めのようにレジスタどうしの内容を転送する場合です．

間接モードとは，同例2，4番めのようにどちらか一方が，データの入っているアドレスを表わしている場合です．2番めの例では1234H番地の内容を転送するという意味であり，4番めの例ではBXレジスタの内容によってアドレスが変化します．BX＝1000Hであるとすれば，1000H番地の内容をAXレジスタに転送します．

なお，後で述べるようにこれらのアドレッシング・モードの呼び方には様々

イミディエイト・モードは数値

間接モードはメモリ

直接モードはレジスタ

な方法があり，国内で発行されている参考書数冊をとってみてもすべて表現がまちまちです．それらの中には非常に理解しにくいものもあり，学習上の障害となっていましたが，本書ではインテル社の表記法をもとに，最も簡潔でわかりやすい方法で解説していますから，本書の表記法を十分に理解すれば他書の表記法の意味もわかるようになるはずです．

以上の説明と多少重複しますが，3種類のアドレッシング・モードについて次にまとめました．

イミディエイト・モード

イミディエイト・アドレッシング・モード

MOV　AX, 1234Hという書き方で，直接数値を代入する方法のことです．これをイミディエイト・アドレッシング・モードといいます．

直接モード（レジスタ・アドレッシング）

レジスタ・アドレッシング・モード

MOV　AX，BXというように，レジスタの内容そのものを転送する方法をレジスタ・アドレッシングといいます．IP，FLレジスタを除くすべてのレジスタが，両方とも16ビットあるいは8ビットである必要があります．したがって，

MOV　AL, BH

は可能ですが，

MOV　AX, BH

のような形は許されません(**MOV命令の整理**　参照)．

間接モード（メモリ・アドレッシング）

メモリ・アドレッシング・モード

間接モードでは転送する値やレジスタではなく，転送する値が格納されているアドレスを指定します．

間接モードでアドレス指定に使用できるレジスタは，BX，BP，SI，DIの4つおよび数値によるアドレス指定です．

これらのレジスタ，および数値の間で様々な組み合わせが可能なために若干複雑にみえる指定方法が現われてきますが，組み合わせ方は単純で規則的ですから恐れることはありません．

一般には**表3.2**が使われていますが，この表より次ページ**表3.3**を覚える方が得策です．disp (dis-placement：変位)は数値であり，8または16ビットの値です．これを符号付数値のように扱い，同一セグメント内の全アドレスを指定できます（**5章-1**参照)．

表3.2　メモリ・アドレッシング一覧表

r/m　mod	00	01	10
000	[BX+SI]	disp8[BX+SI]	disp16[BX+SI]
001	[BX+DI]	disp8[BX+DI]	disp16[BX+DI]
010	[BP+SI]	disp8[BP+SI]	disp16[BP+SI]
011	[BP+DI]	disp8[BP+DI]	disp16[BP+DI]
100	[SI]	disp8[SI]	disp16[SI]
101	[DI]	disp8[DI]	disp16[DI]
110	ダイレクト	disp8[BP]	disp16[BP]
111	[BX]	disp8[BX]	disp16[BX]

mod，r/mの意味については巻末のインストラクションセットを参照

具体的なアドレス指定の方法は次のとおりです．

表3.3

BX BP なし	＋	SI DI なし	＋	disp8 disp16 なし		［　］内からそれぞれひとつ選択する

組み合わせ方はこの表で完璧

たとえば，

BX＋SI

BP＋DI＋disp16

BX＋disp8

disp16

といった組み合わせができるわけです．したがって，3×3×3＝27通りの組み合わせができるわけですが，そのうちのいくつかは存在しません．すなわち，

①　3つとも"なし"の場合は除く．

②　disp 8 単独では 8 ビットの値でありアドレスを指定できないので除く．

③　BP単独は除く（理由は以下で説明）．

の 3 種類です．したがってすべての組み合わせは27－3＝24通りとなります．

また，disp16単独の場合を"ダイレクト"と呼びます．これは，例に示したMOV　AX，［1234H］のような場合で，アドレスの指定を直接数値で行なうモードです．しかしこれは，あくまで「間接モード」のひとつで，他の間接モードがレジスタの値と数値を用いてアドレスを指定しているのに対して，数値のみでアドレスを指定しているという違いがあるだけです．

なお，「イミディエイト・モード」，「直接モード」，「間接モードの中のダイレクト」など，どれも日本語にすれば「直接」という意味ばかりで混同しそうですが，**即値，レジスタ，［　］付き**という区別を明確にして覚えてください．

BPのみの場合を除いたのは，**表3.2**をみるとわかるように，機械語にした場合に表わすことのできる間接指定の組み合わせが24通りしかないためで，縦・横の関係をみるとわかりますが，BPの位置にはかわりに"ダイレクト"が入っています．

本によっては，これらの組み合わせのそれぞれをベース・アドレッシング，インデックス・アドレッシング，ベース・インデックス・アドレッシングなどの名称によって区別していますが，名称で記憶するよりも上の表の形で覚えてしまうのが簡単で確実だと思います．

なお，実際に内蔵モニタのアセンブラで書くときには，

BX＋SI＋1234H　は，

MOV　AX，1234［BX＋SI］

のようにdisp 8 またはdisp16の部分を［　］の前に持ってきます．しかしあくま

間接指定の表記法

でも意味は“BX＋SI＋1234H”番地の内容を，AXレジスタに転送するということです．したがって，

間接指定の表記法

BX＝1000H

SI　＝1111H

であるとすると，前記の場合は，

1234H[BX＋SI]＝[1000H＋1111H＋1234H]＝[3345H]

となるので，3345H番地の内容をAXレジスタに転送するという意味です(**注1**)．

ただし数値のみで指定する場合，すなわち”ダイレクト”のときは，

MOV　AX，[1234]

というように[　]の中に書きます．

注1)　参考書によって，間接指定の[　]の使用法が不統一であるかのように思えるのは，「レジスタ名」と「数値」とが，直接加算できないものである点を考慮したためであろう．レジスタ名はアドレッシングの一種であり，数値そのものではない．そのため，レジスタの内容を表わす場合には，正しくは(AX)のように表記しなければならない．これを用いると，上記の例は“[(BX)＋(SI)＋1234H]”と書かれる．モニタの表記法はこれを略記したものである．本書では特にこのような厳密性は追求せず，モニタの表記法を使用し，レジスタと数値の間でも，直接演算ができるものと考えていく．

2　アドレッシング・モードを用いるときの注意

2-1　間接指定におけるセグメント

BPを含むアドレッシングのときだけSS，その他DS

前項の中で説明し忘れていたことがひとつあります．それは間接指定ではある番地（アドレス）の内容を持ってくるわけですから，その番地（アドレス）がどこのセグメントに属する番地（アドレス）かを明らかにしなければならない，ということです．

結論からいえば，**BPレジスタを含むアドレッシングのときには，スタック・セグメント(SS)から持ってくる，それ以外の場合はすべてデータ・セグメント（DS）から持ってくる，**という約束になっています．これらのセグメントは，先に述べたセグメント・オーバーライド・プリフィックス(CS：，ES：など)を用いることにより変更が可能です．なぜBPレジスタを含むときだけスタック・セグメントになるのかということについては理由があるのですが，これについては，**スタック**の項で簡単に触れます．

2-2　転送先のアドレッシング

転送先のアドレッシング・モードも，転送元とまったく同じです．ただしイミディエイト・アドレッシングがないことはいうまでもありません．

アドレッシングの例としては，

MOV　BX, AX　　：レジスタ・アドレッシング
MOV　[BP+SI], CX：メモリ・アドレッシング

などがあります．

いままで述べてきた場合は少なくとも一方がレジスタでしたが，特殊な場合としては，

MOV　WORD [BX], 1234H

のように，メモリに直接数値を書き込む場合があります．これは転送元にとってはイミディエイト・アドレッシングであり，転送先にとってはメモリ・アドレッシングということになります．その他にも，いくつか特別なものがありますが，詳しくはその都度述べることにします．

例　題

次のアセンブラの記述法のうち，正しいものには○を，誤りのあるものには×をつけ，その理由を示せ．ただし内蔵モニタの形式に従い，すべての数値は16進数であるとみなしてHを省略してある．

① **MOV　[BX], BX**
② **MOV　AX, [CX]**
③ **MOV　[SI], 12**
④ **MOV　[1234], CL**
⑤ **MOV　[BP+BX], AX**
⑥ **MOV　WORD　2345 [BX+DI], 1234**
⑦ **MOV　DS, [BL]**
⑧ **MOV　CS, [DI]**

解　答

① ○
② × CXレジスタは間接指定に使えない．
③ × BYTEデータであることを示さなければならない．
④ ○
⑤ × BPとBXとを同時に組み合わせることはできない．
⑥ ○
⑦ × BLレジスタは8ビットなので間接指定に使用できない．
⑧ × CSレジスタに値を転送することは許されない．なお，内蔵モニタでは，この命令が翻訳可能であるが，実行した場合は暴走する．

練習問題

3—5

1100H番地に直接数値0FFHを代入した上で，これを1200H番地に転送するプログラムを作れ．ただしBXレジスタに1000Hを，SIレジスタに100Hを，DIレジスタに200Hを代入し，これらのレジスタを使った間接アドレッシングを用いるものとする．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 C6060011FF    MOV     BYTE [1100],FF
0009 BB0010        MOV     BX,1000
000C BE0001        MOV     SI,0100
000F BF0002        MOV     DI,0200
0012 8A00          MOV     AL,[BX+SI]
0014 8801          MOV     [BX+DI],AL
0016 F4            HLT
0004 8B37          MOV     SI,[BX]
```

解　説

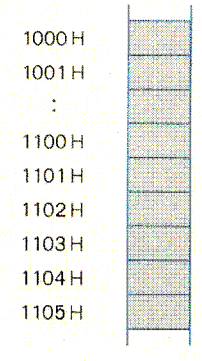
3行め（4H番地）は"ダイレクト"モードで代入していますが，BX，SI，DIレジスタに値を代入したあと，[BX+SI]として指定することもできます．

7，8行目（12H，14H番地）ではALレジスタを用いましたが，当然AH，CLなどを用いることもできます．ただしBH，BLを用いるとアドレス指定が狂ってしまいます．また，AXレジスタを使用することもできますが，あまりおすすめできません．もともとバイトデータとして代入されたものは，バイトデータとして扱いましょう．ただし，バイトデータの並びをワードデータとして扱うように，十分考えられている場合は別です．

3—6

図3.3のようにメモリにデータが入っているとする．このとき次ページの**プログラムA**を実行すると最後に1000H番地に入る数値は何か．ただしDS＝CSとなっているものとする．数値はすべて16進数とする．

図3.3



プログラムA

```
0000 8B1E0010          MOV     BX,[1000]
0006 8B4001            MOV     AX,1[BX+SI]
0009 88260010          MOV     [1000],AH
000D F4                HLT
```

解　答　56H

解　説

順を追って考えていきましょう．1行目でBXに1000H，1001番地の内容が逆ワードで転送され，BX＝1100Hとなります．2行目でSIにBXの示す番地すなわち，1100H，1101H番地の内容が逆ワードで転送され，SI＝0002Hとなります．3行目では，アドレスの計算が面倒ですが，BX＋SI＋01ですから1100H＋0002H＋01H＝1103Hとなり，AXレジスタに1103H，1104H番地の内容が逆ワードで転送されますから（注意！），AX＝5634Hとなります．

4行目でAHのみを1000H番地に転送するのですから，結局56Hが1000H番地に転送されることになります．転送されると，もとからあったデータ(00H)は消えてしまいますから，1000H番地の内容は56Hとなります．

実際に実行するには，次のプログラムを入れてください．結果は見てのとおりです．

```
0000 8CC8              MOV     AX,CS
0002 8ED8              MOV     DS,AX
0004 C706000100011     MOV     WORD [1000],1100
000A C706000110200     MOV     WORD [1100],0002
0010 C606021112        MOV     BYTE [1102],12
0015 C606031134        MOV     BYTE [1103],34
001A C606041156        MOV     BYTE [1104],56
001F C606051178        MOV     BYTE [1105],78
0024 90                NOP
0025 90                NOP
0026 8B1E0010          MOV     BX,[1000]
002A 8B37              MOV     SI,[BX]
002C 8B4001            MOV     AX,01[BX+SI]
002F 88260010          MOV     [1000],AH
0033 F4                HLT

h]G0,33
*1B00:0033
h]D1000
1000 56 11 00 00 00 00 00 00 00 00 00 00 00 00 00 00          V
```

第4章
演算命令を学ぼう

1　加算命令

2　減算命令

3　符号付演算

4　乗除算命令

5　ランダムテクニックとフラグレジスタの働き

1 加算命令

算術演算命令とはその名のとおりCPUに演算をさせる命令のことです．8086では，加減乗除の四則演算を行なうことができます．8ビットのCPUでは，加減算とせいぜい乗算までしか行なうことができなかったのですが，16ビットになって四則演算を行なうことができるようになったため，計算能力は格段に向上しました．さらに8086では，数値演算コ・プロセッサー8087を加えることにより，多桁の加減乗除や関数演算も演算命令として使用できるようになり，飛躍的に計算能力を上げることができるようになりました．

四則演算ができるのは，16ビットの特徴

コ・プロセッサー8087

1 ADD命令

加算命令はその名のとおり，データをたし合わせる命令です．具体的には次のように記述します．

レジスタに数値を加える

```
ADD     AX , 1234H
```

これはAXレジスタに1234Hを加算し，結果をAXレジスタに収めるという意味です．BASICで書くと，次のようになります．

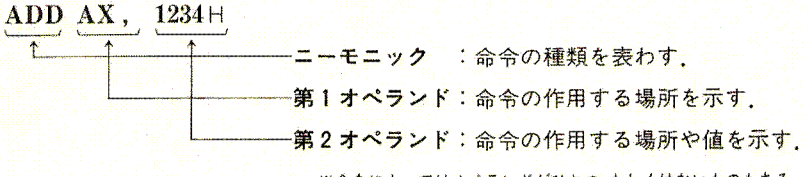
```
AX＝AX＋&H1234
```

加算命令ではすべて第1オペランド（この場合AXレジスタ）に第2オペランド（この場合数値1234H）を加え，結果を第1オペランドに入れます．ここで，復習も兼ねて用語を整理しておきましょう．

第1オペランド

第2オペランド

ニーモニック



※命令によってはオペランドがひとつ，もしくはないものもある．

転送命令（MOV）同様，加算命令（ADD）にも種々のレジスタやメモリの指定法があります．以下にまとめてみましょう．結果はすべて第1オペランドに格納されます．

レジスタ・メモリの指定法の整理

ADD　reg，即値……レジスタに直接数値を加える．
ADD　mem，即値……メモリに直接数値を加える（注）．
ADD　reg，reg ……レジスタにレジスタの値（内容）を加える．
ADD　mem，reg……メモリにレジスタの値（内容）を加える．
ADD　reg，mem……レジスタにメモリの値（内容）を加える．

※この命令群では，結果に従ってフラグが変化する．

注）メモリに直接数値を加える場合には，バイトデータ，ワードデータの区別を明確にするため，BYTEまたはWORDを付ける必要がある．今後直接操作を行なう命令には，注意を喚起する意味で＊印を付けておく．
　レジスタは8ビットまたは16ビットのどちらも使うことができるが，第1オペランドと第2オペランドのタイプが一致していなければならない．どちらか一方がメモリの場合は，自動的にレジスタのタイプに一致するようになる．

例　題

100H番地にバイトデータ11Hを，101H番地にバイトデータ22Hを代入し，この2つのデータの和を102H番地に入れよ．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 C606000111    MOV     BYTE [0100],11
0009 C606010122    MOV     BYTE [0101],22
000E A00001        MOV     AL,[0100]
0011 02060101      ADD     AL,[0101]
0015 A20201        MOV     [0102],AL
0018 F4            HLT
```

←100H番地に11Hを，101H番地に22Hを代入する．

←あらかじめALレジスタに100H番地の内容を入れておき，それに101H番地の内容を加える．

解　説

このプログラムをAコマンドで打ち込み，Lコマンドで逆アセンブルして確認したあと，Gコマンドで実行し，Dコマンドで確認してみます．

11H＋22Hの結果として，33Hが102H番地に入っています．

```
hJG0,18
*1B00:0018
hJD100
0100 11 22 33 00 00 00 00 00 00 00 00 00 00 00 00 00          "3
```

2 桁上がりとキャリーフラグ

前項の例題では，たし算の結果レジスタの大きさを越える桁上がりを生じませんでしたが，桁上がりを生じる場合はどうしたらよいのでしょうか．

桁上がりを生じる場合

2桁の16進数のたし算では，0FFHと0FFHの加算が，考えられる最大の結果を与えます．結果は次のようになります．

16進数		2進数
FF	＝	11111111
＋FF		＋11111111
1FF		111111110

つまり16進数で，2桁めから3桁めに1だけ桁上がりを生じる場合があるわけです．ところが8ビットのレジスタ(たとえばALレジスタ)を使用した加算では，結果は同じ8ビットのレジスタに収められるわけですから，桁上がりがあったとしても，それをレジスタに収納することはできません．

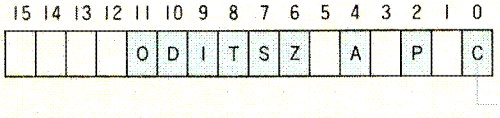
そこでこのような場合のために，フラグ・レジスタの中にキャリーフラグというものが設けられていて，桁上がりのあった場合にはこのフラグを1に，なかった場合には0にすることにして，桁上がりの有無を記憶するようになっています．

フラグ・レジスタ

キャリーフラグ

桁上がりの有無を記憶

図4.1　フラグ・レジスタとキャリーフラグ



フラグ・レジスタは，他のレジスタと違って1ビットずつが別々の意味を持っているので，形式的には1個の16ビットレジスタですが，実際は独立した9個の1ビットレジスタとして働きます(使用されていないビットがある)．フラグレジスタの他のフラグの働きについては，あとでまとめて説明します．

さて，それでは16進数4桁のたし算ではどうなるでしょうか．実際に計算してみると，

2進数		16進数
FFFF	=	1111111111111111
+FFFF	=	+1111111111111111
1FFFF		1 1111111111111110

となり，2桁の場合と同様に1だけ桁上がりが生じます．この場合も計算を実行すると，キャリー・フラグは1にセットされます．AXレジスタを用いて計算したとすれば，AXレジスタに0FFFEHが，CF（キャリー・フラグ）に1が入ります．

図4.2

こうして，あたかもAXレジスタのすぐ隣りにCFがあるかのように計算がなされます．

このような桁上がりがあった場合，2進数で考えて「最上位ビットからの桁上がりがあった」ということもあります．このように，レジスタの働きを考えるときには，ビット単位で考えた方が，わかりやすい場合がよくあります．

ところで2桁のたし算を行なった場合，3桁めに桁上がりを生じるとキャリーフラグは1にセットされましたが，これは8ビットレジスタ（この場合はALレジスタ）を用いて計算を行なったからです．16ビットレジスタを使用して同じ計算をした場合には，キャリーフラグはセットされません．なぜなら2桁のたし算の結果得られた3桁の数は，十分にAXレジスタに収納できるからです．

演算に使用するレジスタで桁上がりの有無が異なる

このように，**キャリーフラグは使用するレジスタが8ビットであっても16ビットであっても，そのレジスタに結果が収納しきれなくなった場合にのみ，1にセットされます．**そうでない場合には常に0にリセットされます（注1）．

したがって8ビットレジスタを用いるか16ビットレジスタを用いるかによって，結果の収納のされ方が違ってきます．

セット
リセット
クリア

例　**AX＝00FFHのとき（つまりAL＝FFH），これに0FFHを加える操作を考える．**

ADD　AL，0FFH　とした場合⟶

AX	
AH	AL
00	FE

CF
1

ADD　AX，00FFH　とした場合⟶

AX	
AH	AL
01	FE

CF
0

注1）　あるビットの値（この場合はキャリーフラグ）を1にすることをセットするといい，0にすることをリセットするという．0にすることをクリアするという場合もある．

3 ADC命令

8086は16ビットのCPUであり，すべてのレジスタが16ビット長です．それでは16ビット以上の数値の計算をするときには，どうしたらよいのでしょうか．

16ビットのレジスタを用いた計算では，一度に16ビットまでしか計算することができません．そこで16ビットを越える数値を扱う場合には，何度かに分けて計算することになります．たとえば，次のような32ビットの加算を行なう場合です．

0FFFFFFFH＋0FFFFFFFH＝1FFFFFFEH

```
      [1] ←─────┐
   0FFF         │   FFFF
 ＋0FFF         │ ＋FFFF
 ──────         │ ──────
   1FFF         └[1]FFFF
```

ADC命令は桁上がりを同時に加える

まずはじめに，下位の16ビット（16進４桁）の加算を行ないます．その結果１だけ桁上がりが生じます．この桁上がりは加算の結果キャリーフラグにセットされているので，上位16ビット（４桁）の加算を行なうときにその結果に加えてやればよいことになります．このことは普段私たちが10進数のたし算を行なうときに，１桁ずつ桁上がりをしていることと同じですから別に難しい考え方ではありません．16ビット計算では，４桁に１回だけ考えればよいという点が違うだけです．

10進数加算の例

```
   4 8 7 5
   2 9 1 6
 + 1 0 1      ←── 桁上がり
 ─────────
   7 7 9 1
```

キャリーフラグの内容は計算結果で変化する

このような加算を行なうためには，上の例では上位16ビットを計算するときに通常の加算を行なったあと，キャリーフラグを加えてやればよいことがわかります．計算のたびにキャリーフラグを加えるのは面倒ですので，そのためにADC（Add with Carry）命令が用意されています．この命令を用いることにより，通常の加算結果にキャリーフラグが自動的に加算されます．この場合も計算の結果さらに桁上がりを生じれば，キャリーフラグは１にセットされます．つまり**新たな計算をすると前のキャリーフラグの内容は失われ，新しい計算の結果に従ってセットあるいはリセットされます．**

下位16ビットの計算の結果，桁上がりを生じない場合はキャリーフラグは０

にリセットされていますから，上位16ビットの計算の際にADC命令を用いても何の影響も及ぼさないことになります．したがって，例のような32ビット演算では桁上がりの有無にかかわらず，ADC命令によって正しい答を得ることができるわけです．

ADC命令の整理

ADC　reg，即値……レジスタに直接数値を加え，さらにCFの値を加える．
ADC　mem，即値　＊……メモリに直接数値を加え，さらにCFの値を加える．
ADC　reg，reg……レジスタにレジスタの値（内容）を加え，さらにCフラグの値を加える．
ADC　mem，reg……メモリにレジスタの値（内容）を加え，さらにCフラグの値を加える．
ADC　reg，mem……レジスタにメモリの値（内容）を加え，さらにフラグの値を加える．

※この命令群では結果に従ってフラグが変化する．

例題1

56789ABH＋8888888Hを求め，結果の上位16ビットをDXレジスタに，下位16ビットをAXレジスタに入れよ．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 B8AB89        MOV     AX,89AB
0007 BA6705        MOV     DX,0567
000A 058888        ADD     AX,8888
000D 81D28808      ADC     DX,0888
0011 F4            HLT
```

AXレジスタが下位16ビット，DXレジスタが上位16ビットであることに注意．

下位のAXレジスタを先に計算し，上位のDXレジスタにはADC命令を用いる．

解　説

AX，DXレジスタのどちらが上位か下位かをよく考えて，プログラムを作ります．はじめに下位16ビットの加算を行なう場合には，キャリーフラグの内容を加える必要がないので，ADD命令を用います．この場合は以前の計算結果によってキャリーフラグがどうなっているのかわからないので，ADC命令を用い

下位の加算はADD命令

てはなりません．上位ビットの計算にはADC命令を用います．

結果を確かめてみましょう．

```
h]G0,11⏎
*1B00:0011
```

としたあとで，Xコマンドでレジスタの内容を調べます．

```
h]X⏎
FL ------AP- ←―― フラグレジスタの見方については
AX 1233           あとで説明します．
BX 000A ←┐
CX 25EF ←┴―― BX，CXレジスタは今回の計算には
DX 0DF0          関係がないので，必ずしもこの値に
SP F7F6          なっている必要はない．
BP 04F6
SI 0000
DI 0000
CS 1B00
DS 1B00
SS 0060
ES 0000
IP 0011
```

つまり，

AX＝1233H

DX＝0DF0H

ということですから，加算結果は，

56789ABH ＋**8888888**H ＝**0DF01233**H

となります．

例題２

> 前問の結果を100H番地から順に格納せよ．その際100H番地が最上位のバイト……103H番地が最下位のバイト，となるようにせよ．

解　答

```
0000 8CC8      MOV   AX,CS     ┐
0002 8ED8      MOV   DS,AX     │
0004 B8AB89    MOV   AX,89AB   │←―ここまでは前問と同じ．
0007 BA6705    MOV   DX,0567   │
000A 058888    ADD   AX,8888   │
000D 81D28808  ADC   DX,0888   ┘
0011 86E0      XCHG  AH,AL ←―――― AHとALの内容を入れ換える命令．
0013 86F2      XCHG  DH,DL ←―――― DHとDLの内容を入れ換える．
0015 89160001  MOV   [0100],DX
0019 A30201    MOV   [0102],AX
001C F4        HLT
```

解　説

練習問題3-4（P.86）で，逆ワード形式にならないようにデータを格納する方法を考えました．この場合はAL，AHを単独のレジスタとして扱って転送しましたが，ここではあらかじめALレジスタとAHレジスタの内容をXCHG命令を使って入れ換えてから，AXレジスタとして一括して転送を行なっています．では，次項でこの命令について詳しく解説しましょう．

4 XCHG命令

XCHG（eXCHanGe）命令は，レジスタどうし，あるいはメモリとレジスタの内容を交換する命令です．これを用いることにより，第3のレジスタを使用することなく内容を交換することができます．

内容を交換する命令

使用できるのは8ビット，16ビットのレジスタあるいはメモリで，第1オペランドと第2オペランドのタイプが一致している必要があります．BASICのSWAP命令と同じ働きと考えればよいでしょう．

XCHG命令の整理

XCHG　reg，reg ……レジスタどうしの内容を交換する．
XCHG　mem，reg ……メモリとレジスタの内容を交換する．

※この命令群ではフラグは変化しない．

当然のことですが，

```
XCHG　BX，CX
```

と，

```
XCHG　CX，BX
```

とはまったく同じ動作をしますし，また，

```
XCHG　[100]，BX
```

と，

```
XCHG　BX，[100]
```

ともまったく同じ動作をします．場合によって生成する機械語コードが違うことがありますが，動作は同じです．あとの例の場合，Aコマンドでの入力のときはどちらでもかまいませんが，Lコマンドで逆アセンブルすると両方とも，

```
XCHG　[1000]，BX
```

の形になります．

8086では同じ命令に対して複数の機械語コードが考えられる場合がありますが，これはプログラム上での柔軟性を持たせるためで，動作はまったく同じになります．

ここで，先の**例題2**の実行結果をみてみましょう．

```
hJG0,1C
*1B00:001C
hJD100
0100 0D F0 12 33 00 00 00 00 00 00 00 00 00 00 00 00          × 3
```

となっていることがわかります．

例題2の解答と同じはずで，答は0DF01233Hとなっています．

練習問題

4—1

100H，101H番地にワードデータ1111Hを，102H，103H番地にワードデータ2222Hを代入し，2つのデータの和を104H，105H番地に入れよ．ただしアドレス指定にはBXレジスタを用いた間接指定法を用いること．

解答

```
0000 8CC8            MOV     AX,CS
0002 8ED8            MOV     DS,AX
0004 BB0001          MOV     BX,0100
0007 C7071111        MOV     WORD [BX],1111
000B C747022222      MOV     WORD 02[BX],2222
0010 8B07            MOV     AX,[BX]
0012 034702          ADD     AX,02[BX]
0015 894704          MOV     04[BX],AX
0018 F4              HLT
```

メモリに直接数値を書き込むときは，BYTEまたはWORDを指定する．

例題を参考に，ALレジスタのかわりにAXレジスタを用いる．

解説

間接指定法は大丈夫ですか．例題と違っているのは，使用するレジスタが16ビットレジスタである点ですね．

では実行してみましょう．

```
hJG0,18
*1B00:0018
hJD100
0100 11 11 22 22 33 33 00 00 00 00 00 00 00 00 00 00          ""33
                  ──結果
```

結果は3333Hとなっています．なお，結果は逆ワード形式になっているはず

ですが，この問題の場合はたまたま上位，下位のバイトとも33Hなので，一見何も問題がないようにみえるのです．

4−2

56789ABHを100H番地以後に，8888888Hを110H番地以後に格納し，その和を120H番地以後に格納せよ．ただし計算にはAXレジスタのみを用い，結果は逆ワード形式で格納されてもよいものとする．またBXレジスタに100Hを，SIレジスタに10Hを，DIレジスタに20Hを代入し，これらのレジスタを用いた間接アドレス指定法を用いる．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 BB0001        MOV     BX,0100
0007 BE1000        MOV     SI,0010
000A BF2000        MOV     DI,0020
000D C7076705      MOV     WORD [BX],0567
0011 C74702AB89    MOV     WORD 02[BX],89AB
0016 C7008808      MOV     WORD [BX+SI],0888
001A C740028888    MOV     WORD 02[BX+SI],8888
001F 8B4702        MOV     AX,02[BX]
0022 034002        ADD     AX,02[BX+SI]
0025 894102        MOV     02[BX+DI],AX
0028 8B07          MOV     AX,[BX]
002A 1300          ADC     AX,[BX+SI]
002C 8901          MOV     [BX+DI],AX
002E F4            HLT
```

上位4桁を100H番地以後に，下位4桁を102H番地以後に格納する．

上位4桁を110H番地以後に，下位4桁を112H番地以後に格納する．

下位4桁を先に加算する．キャリーフラグはMOV命令によって変化しないので，前の演算結果（0022H番地）の桁上がりを保持している．

解　説

アドレスの指定に間接指定を用いているので，なかなかやっかいなプログラムだったと思います．解答を見ずにこのプログラムが書けるようであれば，いままで学習してきたことが十分に理解されているといえるでしょう．もしできなかったとしても心配することはありません．今後練習を重ねることによって，理解できるようになるでしょう．

CFはMOV命令では変化しない

ここで用いたBX，SI，DIのレジスタにはそれぞれ意味があります．すなわちBXはベースレジスタ，つまりデータの格納される最初の場所を示しています．SIは**ソースインデックス**，すなわち**もとになるデータ**（この場合は加算するデータ）があるアドレスがベースレジスタからどれだけ離れているかを示しています．DIは**ディスティネーションインデックス**，すなわち目的のデータ（この場合は加算結果）を**収納するアドレス**がベースレジスタからどれだけ離れているかを示しています．

ベースレジスタを基本にソースインデックス・ディスティネーションインデックスがそこからの距離

このように，レジスタの名称とその使われる意義を同時に理解すれば，複雑な間接アドレス指定方式も決して難しいものではなくなるはずです．

では，このプログラムを実行してみましょう．

```
h]G0,2E
*1B00:002E
h]D100,12F
0100 67 05 AB 89 00 00 00 00 00 00 00 00 00 00 00 00        g ｫ|
0110 88 08 88 88 00 00 00 00 00 00 00 00 00 00 00 00        | ||
0120 F0 0D 33 12 00 00 00 00 00 00 00 00 00 00 00 00        × 3
```

となっていることがわかります．0120番地からが結果で，逆ワード形式で格納されていますから，正しい答は，

0DF01233H

となり，前問までの解答と一致していることがわかります．

2 減算命令

1 SUB命令

減算命令でも加算命令と同じように第1オペランドから第2オペランドを減じ，その結果を第1オペランドに格納します．

減算はSUB（SUBtract）命令を用います．以下にその使用法をまとめてみましょう．

SUB命令の整理

SUB　reg，即値 ……レジスタから直接数値をひく．
SUB　mem，即値　＊……メモリから直接数値をひく．
SUB　reg，reg ……レジスタからレジスタの値（内容）をひく．
SUB　mem，reg ……メモリからレジスタの値（内容）をひく．
SUB　reg，mem……レジスタからメモリの値（内容）をひく．

※この命令群では結果に従ってフラグが変化する．

例　題

100H，101H番地にワードデータ2345Hを，102H，103H番地にワードデータ1111Hを代入し，100H，101H番地の内容から102H，103H番地の内容を減じた答を，104H，105Hに格納せよ（データはすべて逆ワード形式とする）．

解　答

```
0000 8CC8            MOV     AX,CS
0002 8ED8            MOV     DS,AX
0004 C70600014523    MOV     WORD [0100],2345
000A C70602011111    MOV     WORD [0102],1111
0010 A10001          MOV     AX,[0100]
0013 2B060201        SUB     AX,[0102]
0017 A30401          MOV     [0104],AX
001A F4              HLT
```

AXレジスタから102H，103H番地の内容を減じ，結果をAXレジスタに残す．

解　説

使用法は基本的には加算命令と同じです．え？　結果が負になったらどうするかって？　まあ，あせらず読み進めていってください．順序だてて説明していきますから…．あせりは禁物です．

では，正しく実行されるかどうか確かめてみましょう．

```
hJG0,1A
*1B00:001A
hJD100
0100 45 23 11 11 34 12 00 00 00 00 00 00 00 00 00 00          E#  4
```

となり，104H，105H番地に逆ワード形式で1234Hが格納されているのがわかります．

2 SBB命令

SBB（SuBtract with Borrow）命令も，加算命令におけるADC命令と同様な働きをします．つまり多桁の減算命令を実行する場合に，上位のバイトに対して使用されます．

"借り"とキャリーフラグ

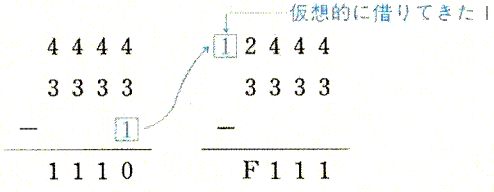
減算命令の場合は，演算の結果ひくことができないときには上位の桁から借りてくることによって，減算を実行します．この場合も確かめてみればわかることですが，上位の桁から借りてくるのは，最大限1に限られます．

そこで，この場合も“借り”が生じた場合，キャリーフラグを1にセットします．この点で本来なら“Borrow”というべきなのでしょうが，煩雑になることを避けるために，キャリーフラグを共用しています．キャリーフラグは加減算の他にもいくつかの命令で使用されていますが，直前に実行された命令の種類

によってその意味が変わってきます．しかしキャリーフラグを使う命令は，ADC命令のように前の命令の直後に置かれるので，混乱が生じることはありません．

ともかく減算の場合は，上位の桁から“借り”が生じたときにセットされるので，上位の桁を計算する際にはその貸した分もひかなくてはなりません．そのための命令がSBB命令で，通常の減算命令を実行後キャリーフラグの値をひきます．この場合も借りがなければSUB命令と同じ働きになるわけですから，完全に正しい答を得ることができます．なお，減算結果が負の値になる場合については次の節で説明します．

SBB命令の実行例



SBB命令の整理

SBB　reg，即値……レジスタから直接数値をひき，さらにCFの値をひく．
SBB　mem，即値　＊……メモリから直接数値をひき，さらにCFの値をひく．
SBB　reg，reg……レジスタからレジスタの値(内容)をひき，さらにCFの値をひく．
SBB　mem，reg……メモリからレジスタの値(内容)をひき，さらにCFの値をひく．
SBB　reg，mem……レジスタからメモリの値(内容)をひき，さらにCFの値をひく．

※この命令群では結果に従ってフラグが変化する．

練習問題

4−3

上の計算式を実行するプログラムを作れ．**練習問題4-2**と同様に100H番地以後に44442444Hを，110H番地以後に33333333Hを，120H番地以後に結果を入れること．BX＝100H，SI＝10H，DI＝20Hとする．

解答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 BB0001        MOV     BX,0100
0007 BE1000        MOV     SI,0010
000A BF2000        MOV     DI,0020
```

```
000D C7074444        MOV     WORD [BX],4444
0011 C747024424      MOV     WORD 02[BX],2444
0016 C7003333        MOV     WORD [BX+SI],3333
001A C740023333      MOV     WORD 02[BX+SI],3333
001F 8B4702          MOV     AX,02[BX]
0022 2B4002          SUB     AX,02[BX+SI]
0025 894102          MOV     02[BX+DI],AX
0028 8B07            MOV     AX,[BX]
002A 1B00            SBB     AX,[BX+SI]
002C 8901            MOV     [BX+DI],AX
002E F4              HLT
```

前問同様，MOV命令ではキャリーフラグは変化しないので，前の減算の結果生じたボローが次の減算のときにあわせてひかれる．

解　説

練習問題4-2を参考にすれば，数値を変え，ADD命令をSUB命令に，ADC命令をSBB命令に変えるだけですむことがわかると思います．

間接アドレス指定は変更が容易

みなさんの中には，どうしてわざわざBXレジスタやSIレジスタ，DIレジスタを使って番地を指定するのだろうか，と思われた方もいるかも知れません．何度もBX，SIなどを書くよりも100H番地，110H番地などと書いた方が直接的でわかりやすいのにと考える方もいるでしょう．

しかし，このように番地をあるレジスタで指定することには，大きなメリットがあります．それはデータを収納するアドレスを変更する必要が生じたときに，はじめにレジスタに値を入れる部分だけ書き直せばよい，という点です．

このような方法は，リロケータブル（位置自由）なプログラムを書く上でも必要とされる技術なので，早めに修得するようにしてください．結果は，

```
hJG0,2E
*1D00:002E
hJD100,12F
0100 44 44 44 24 00 00 00 00 00 00 00 00 00 00 00 00     DDD$
0110 33 33 33 33 00 00 00 00 00 00 00 00 00 00 00 00     3333
0120 10 11 11 F1 00 00 00 00 00 00 00 00 00 00 00 00       円
```

となっているはずです．

3 符号付演算

1 補数表現と加減算

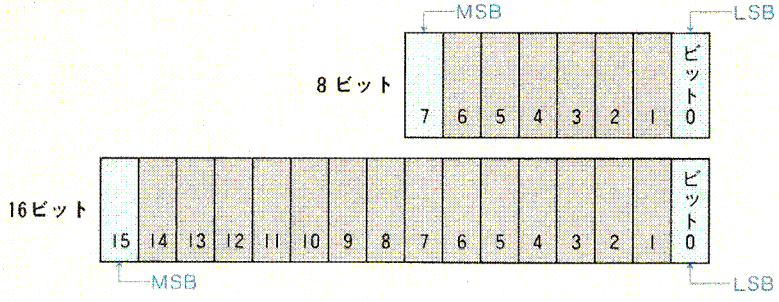
負の数の扱い方

ところでいままで扱ってきた数は，すべて正の数として考えてきました．しかしコンピュータで扱う数は，正の数に限られているわけではありません．負の数を扱う場合もあるでしょうし，何より減算の結果が負になることはいくらでもあるわけですから，負の数の扱い方を決めておかなければなりません．では，実際にどのように扱ったらよいのでしょうか．

2の補数

「2進数と16進数」のところで，コンピュータでは負の数を表現する場合2の補数を用いる，と説明しました．覚えていますか．忘れてしまった人は先へ進む前にもう一度読み直してください．

図4.3



この2の補数表現を用いることにより，正負の数を含めた演算が可能になります．2の補数表現を用いる場合，最上位ビット（MSB）が1のとき負の数を表わすわけですから，8ビットの16進数では0H～7FHまでが正の数を，80H

～FFHまでが負の数を（－80H～－1Hに対応）表わすことになりますし，16ビットの16進数では0H～7FFFHまでが正の数を，8000H～FFFFHまでが負の数を（－8000H～－1Hに対応）表わしています．**2の補数表現による数は，加減算の結果が同じ数の範囲（8ビットなら－80H～7FH）になる限り，正しい結果を表現することができます．**

なおここで忘れてならないことは，数値を正の数として扱うか2の補数として扱うかはプログラム作成する人が決めることであって，機械にはそれを判別する能力はない，ということです．0FFHが＋255dのことなのか，－1dのことなのかは機械にはわかりませんから，プログラムを作る人がしっかりと判別できていないと困ります．このことは，あとで説明する符号付演算命令では特に重要なことです．結果が範囲を越える場合については，次項で説明します．

補数表現の使用を決めるのはプログラマー

例　題

33Hを100H番地に，－22Hを101H番地にそれぞれ代入し，その和を102H番地に，差を103H番地に入れよ．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 C606000133    MOV     BYTE [0100],33
0009 C6060101DE    MOV     BYTE [0101],100-22
000E A00001        MOV     AL,[0100]
0011 02060101      ADD     AL,[0101]
0015 A20201        MOV     [0102],AL
0018 A00001        MOV     AL,[0100]
001B 2A060101      SUB     AL,[0101]
001F A20301        MOV     [0103],AL
0022 F4            HLT
```

←アセンブラ中に計算式を入れることができる．詳しくは解説を参照．

解　説

負の数を表わすためには，補数の計算をしなければなりません．しかしこれは意外と面倒なものです．本例題のように－22Hを表わすためには2^8すなわち100Hから22Hをひかなければなりません．この計算を表や手計算でやっていたのでは大変です．そもそもコンピュータは“計算のための機械”なのですから，このようなことはコンピュータがやるのが当然なのです．

モニタで負の数を入力する方法

というわけで，PC-9801の内蔵アセンブラではこの例のように，アセンブラ中に計算式を書くことができます．この場合は－22Hの部分は100H－22Hとすればよいわけです．結果が0DEHとして格納されているのがわかりますね．ただし使用できる演算子は＋－だけに限られています．

計算結果は次のようになります．

```
h]G0,22
*1D00:0022
h]D100
0100 33 DE 11 55 00 00 00 00 00 00 00 00 00 00 00 00        3^ U
```

2 8ビットデータの16ビットへの変換

符号付演算には，8ビットの場合には結果が−128d～＋127dの間に，16ビットの場合には−32768d～＋32767dの間になければ正しい答が得られない，という制限がついていました．このような制限があると実用上大変不便です．8ビットの2数を加えただけでも制限を越えてしまう場合があるからです．

多桁の符号付演算

符号なしの加減算の場合には，キャリーフラグを使って多桁の計算をすることができましたが，符号付演算の場合には，簡単にキャリーフラグを使って多桁の計算をすることはできません．それは何度も述べているとおり，最上位ビット（MSB）が数値ではなく符号を表わしているからです．

それでは，いったいどうすれば符号付演算の正しい結果が得られるのでしょうか．その方法について説明する前に，次の**例題**を見てください．

例　題

符号なし数値77Hを100H番地に，0AAHを101H番地に，0FFHを102H番地にそれぞれ代入し，3数の和を110H，111H番地に格納せよ（逆ワード形式でよい）．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 C606000177    MOV     BYTE [0100],77
0009 C6060101AA    MOV     BYTE [0101],AA
000E C6060201FF    MOV     BYTE [0102],FF
0013 B80000        MOV     AX,0000
0016 BB0000        MOV     BX,0000
0019 A00001        MOV     AL,[0100]
001C 8A1E0101      MOV     BL,[0101]
0020 01D8          ADD     AX,BX
0022 8A1E0201      MOV     BL,[0102]
0026 01D8          ADD     AX,BX
0028 A31001        MOV     [0110],AX
002B F4            HLT
```

AX，BXレジスタを加算に使用するので，あらかじめクリアしておく．

8ビットデータを16ビットレジスタに加算するため，BLレジスタとBXレジスタを使用している．

解　説

解答をみて，何か変なことをしているなと思われた方は，脈があります．前述したとおり，8ビットの数値を加算すると桁上がりを生じる場合がありますから，得られる答は16ビットになります．前述の方法を用いれば，次のようなプログラムになるでしょう．

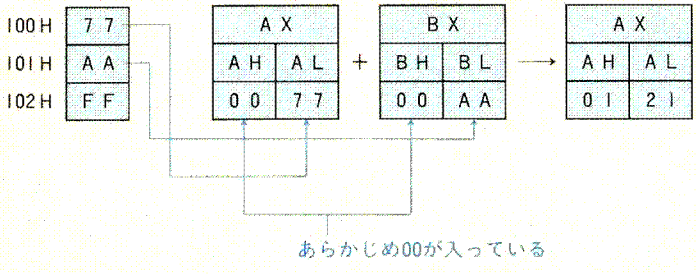
```
0000 A00001          MOV     AL,[0100]
0003 02060101        ADD     AL,[0101]
0007 80D400          ADC     AH,0000
```

← CFの内容のみをAHレジスタに加えるためには，このようにする．

この例では8ビットレジスタALに8ビットの数値を加え，桁上がり分のみをAHレジスタに積算しています．しかし，どうせ加算の結果には16ビットのレジスタを使わなければならないのなら，はじめから16ビットのレジスタを使って加算していけば，キャリーフラグを使わずにすむのではないか，と考えて作ったのが解答に示したプログラムです．

このプログラムでは，あらかじめ8ビットの数値を16ビットに変換してから，16ビットレジスタどうしの加算を行なっています．つまり，あらかじめAXレジスタを0000Hにクリアしておき，その下位8ビットであるALレジスタに，加算で使用する8ビットデータを転送します．すると77Hという8ビットの数値が，0077Hという16ビットの数値として扱えることになります．BXレジスタについても同様に行ないます(**図4.4**参照)．というのは16ビット＋8ビットというように，違う大きさのレジスタでの加減算はできないからです．

図4.4



どうしてわざわざBXレジスタを使って，こんな面倒なことをするのか，001CH番地のプログラムを，

ADD　AX,[0101]

としたらいいじゃないか，と考えた人もいるかも知れませんが，これではダメです．なぜでしょうか？

データのタイプに注意

この命令ではAXレジスタに，101H番地と102H番地の内容が加えられてしまいます．レジスタがワードタイプの場合は，データは自動的にワードタイプと

して解釈されるということを思い出してください．102H番地には別のデータが入っているので，この方法では正しい答は得られません．もし，どうしてもこの方法を使いたいときには，あらかじめデータを2バイトおきに代入し，空白のバイトには00Hを入れておくなどの方法をとらなければなりません．興味のある方は，この方法でプログラムを組んでみてください．

このように8ビットの数値を16ビットに拡張して計算することにより，256個(＝100H個）までのデータの加減算を，オーバフローせずに行なうことができます．

では，結果を確認してみましょう．

```
hJG0,2B
*1D00:002B
hJD100
0100 77 AA FF 00 00 00 00 00 00 00 00 00 00 00 00 00    wｪ
0110 20 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

結果は0220Hとなっていますね．

3 符号付演算と符号拡張命令

それではいよいよ符号付演算で正しい結果を得る方法について，説明をはじめましょう．

符号ビットの移動

例題(P.113)でわかったことは“計算の際，8ビットの数値が多い場合は，あらかじめ16ビットにしておくとよい”ということです．符号付演算でも，基本的には同じことがいえます．しかし，この方法をそのまま使用することはできません．なぜでしょうか？

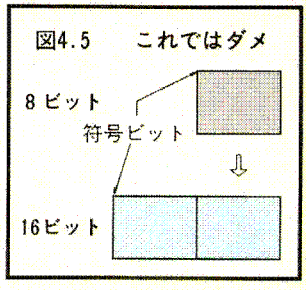


図4.5　これではダメ

8ビットの符号付数値では，0H～7FHが正の数を，80H～0FFHが負の数を表わす，ということを前の方で学習しました．すなわち，最上位ビット（MSB）が1となっている数値は，負の数を表わすということでした．したがって8ビットの数値の場合にはビット7が符号を表わし，16ビットの数値の場合にはビット15が符号を表わしています．

そこで8ビットの符号付数値を，16ビットの符号付数値にするためには，符号の場所を動かさなければならないことに気付くでしょう．しかし8ビットの数値を16ビットに変換するためには，単に符号の場所を変えただけではダメなのです．例として，－1を表わす8ビットの表現と，16ビットの表現を考えてみましょう．

▶－1の表現法◀

	16進数		2進数
8ビット	0FFH	＝	11111111
16ビット	0FFFFH	＝	1111111111111111

符号ビット(bit7)

符号ビット(bit15)

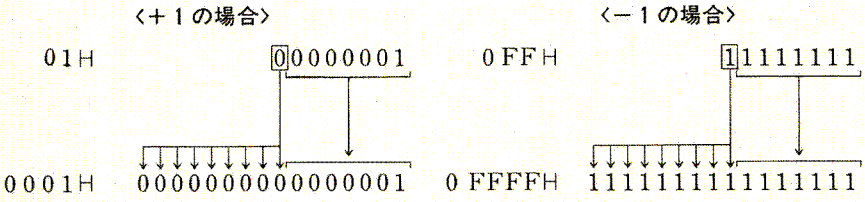
このように，符号ビット以外のビットにも変化が起きているのがわかります．結論からいうと，8ビットの負数を16ビットで表わすときには，上位バイトに0FFHを入れればよいのです．つまり8ビットの80H～0FFHは，16ビットの0FF80H～0FFFFHに対応するわけです．

8ビット符号付数値を16ビットにすると上位バイトは0Hか0FFH

正の数については上位バイトに00Hを入れるだけでよく，8ビットの00H～7FHは，16ビットの0000H～007FHに対応します．このような方法で8ビットの符号付数値を16ビットの符号付数値に変換することを，“符号拡張”と呼び，これを自動的に行なう命令としてCBW（Convert Byte to Word）が用意されています．

符号バイトの拡張

この命令を実行すると，8ビット数値の符号ビットが0のときは16ビットの上位バイトのすべてに0が，符号ビットが1のときには上位バイトのすべてに1が入ります．



つまり，みかけ上符号ビットが上位バイトに拡張されたように見えるため，この命令のことを“符号拡張命令”というわけです．

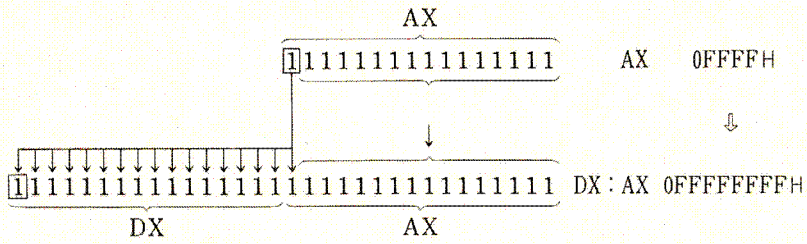
符号拡張命令は，AL，AXレジスタ専用

なお，この命令はALレジスタの符号付バイトデータを，AXレジスタの符号付ワードデータに直す場合の専用命令です．他のレジスタには使用できませんので注意が必要です．

この命令を用いて8ビットの符号付数値を16ビットの符号付数値に符号拡張することにより，256個（＝100H個）までのデータの加減算でオーバーフローせずに結果を得ることができます．

8086ではこの他に，16ビットの符号付数値を32ビットの符号付数値に符号拡張する命令として，CWD（Convert Word to Double word）が用意されています．この命令では，DXレジスタとAXレジスタを1組として，DXレジスタが上位16ビットを，AXレジスタが下位16ビットを表わすものと考えて，AXレジスタの符号付16ビット数値をDXレジスタとAXレジスタによる32ビット符号付数値に符号拡張します．

多数のデータの加減算を行なう方法



この命令を用いて16ビットの符号付数値を32ビットの符号付数値に符号拡張することにより，65536個（＝10000H個）までのデータの加減算でオーバーフローせずに結果を得ることができます．ただしこの場合は，DXレジスタとAXレジスタの32ビットデータを一度に加算または減算することはできません．まず先にAXレジスタを計算し，そのキャリーフラグをDXレジスタに合わせる点に注意してください．

符号拡張命令の整理

CBW ……ALレジスタの8ビット符号付数値をAXレジスタの16ビット符号付数値に変換．

CWD ……AXレジスタの16ビット符号付数値を，DXレジスタを上位16ビット，AXレジスタを下位16ビットとみた仮想的32ビットレジスタの32ビット符号付数値に変換．

※この命令群でフラグは変化しない．

例　題

8ビット符号付数値11Hを100H番地に，88Hを101H番地に，99Hを102H番地に代入しその和を110H，111H番地に格納せよ(逆ワード形式でよい)．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 C606000111    MOV     BYTE [0100],11
0009 C606010188    MOV     BYTE [0101],88
000E C606020199    MOV     BYTE [0102],99
0013 A00001        MOV     AL,[0100]
0016 98            CBW
0017 89C3          MOV     BX,AX
0019 A00101        MOV     AL,[0101]
001C 98            CBW
001D 01C3          ADD     BX,AX
001F A00201        MOV     AL,[0102]
0022 98            CBW
0023 01C3          ADD     BX,AX
0025 891E1001      MOV     [0110],BX
0029 F4            HLT
```

この1命令でALレジスタの内容を拡張して，AXレジスタに入れることができる．

結果はBXレジスタに順に加算していく．

2つ目の数を加える．

3つ目の数を加える．

解　説

CBW命令はみてわかるとおり，オペランドを持たない命令です．この命令を実行すると，ALレジスタの内容がAXレジスタに符号拡張されて入ります．もともとAHレジスタに入っていたデータは失われるので，注意が必要です．結果を順にBXレジスタに加算していくと，16ビットの加算結果が得られます．

AHレジスタ内の内容は失なわれるので注意

実行すると，次のようになります．

```
hJG0,29
*1D00:0029
hJD100,11F
0100  11 88 99 00 00 00 00 00 00 00 00 00 00 00 00 00        ┐
0110  32 FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00        2
```

加算結果が0FF32Hであることがわかります．みなさんの中に，わざわざ16ビットにする必要はなかったのではないか，と考えた人はいないでしょうね．落ち着いてよく考えてみてください．上位バイトが0FFHですから，この数は負の数です．8ビットではこの数値を表現することはできません．

練習問題

4−4

前ページの**例題**において，結果をBXレジスタに加算せずに，直接メモリに加算する方法でプログラムを作れ．

解　答

```
0000 8CC8              MOV     AX,CS
0002 8ED8              MOV     DS,AX
0004 C606000111        MOV     BYTE [0100],11
0009 C606010188        MOV     BYTE [0101],88
000E C606020199        MOV     BYTE [0102],99
0013 A00001            MOV     AL,[0100]
0016 98                CBW
0017 A31001            MOV     [0110],AX
001A A00101            MOV     AL,[0101]
001D 98                CBW
001E 01061001          ADD     [0110],AX
0022 A00201            MOV     AL,[0102]
0025 98                CBW
0026 01061001          ADD     [0110],AX
002A F4                HLT
```

解　説

あえて解説を付ける必要もないと思います．BXレジスタを使用している部分に，メモリを指定すればよいだけです．実行結果は前問と同じですので，自分

で確認してください．それではこの項の仕上げとして，次の**練習問題**をやってみてください．けっこう手応えのある問題ですから，よく考えてください．

4—5

次の16ビット符号付データどうしの加減算を行なえ．なお番地指定にはBXレジスタを用いた間接指定を用い，データは100H番地以後の2バイトずつに，結果は110H番地以後に格納する（逆ワード形式でよいものとする）．

1234H　＋　8888H　－　7777H

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 BB0001        MOV     BX,0100
0007 C7073412      MOV     WORD [BX],1234
000B C747028888    MOV     WORD 02[BX],8888
0010 C747047777    MOV     WORD 04[BX],7777
0015 8B07          MOV     AX,[BX]
0017 99            CWD
0018 894710        MOV     10[BX],AX
001B 895712        MOV     12[BX],DX
001E 8B4702        MOV     AX,02[BX]
0021 99            CWD
0022 014710        ADD     10[BX],AX
0025 115712        ADC     12[BX],DX
0028 8B4704        MOV     AX,04[BX]
002B 99            CWD
002C 294710        SUB     10[BX],AX
002F 195712        SBB     12[BX],DX
0032 F4            HLT
```

AXレジスタの16ビットデータをDXとAXレジスタの32ビットデータに符号拡張する．

あらかじめ，結果を格納する番地を決めておき，データを転送する．

32ビットデータをメモリに直接加算する．キャリーフラグを加えることを忘れないこと．

加算と同様，キャリーフラグの処理を忘れないこと．

解　説

16ビット符号付の加減算を行なうためには，32ビットに符号拡張しなければなりません．32ビットのデータをレジスタどうしで加減算するのは困難ですから，あらかじめ結果を格納する番地を決めておき，直接メモリに加算する形式をとるのが効率がよいでしょう．結果は次のようになります．

```
h]G0,32
*1D00:0032
h]D100,11F
0100  34 12 88 88 77 77 00 00  00 00 00 00 00 00 00 00    4 ll ww
0110  45 23 FF FF 00 00 00 00  00 00 00 00 00 00 00 00    E#
```

逆ワード形式で格納され，しかも下位16ビットが110H番地以後に，上位16ビットが112H番地以後に格納されているので，結果は0FFFF2345Hとなります．この数は負の数で，－0DCBBHのことです（2^{32}の補数です）．

4 乗除算命令

加算命令，減算命令の項にかなり紙面をさきましたが，いままで学習してきたことは機械語を理解するための基礎となることばかりなので，あえて十分過ぎるくらいの説明をしたわけです．これまでのことが理解されていれば，これから解説していく乗算命令や除算命令は容易に理解できるはずです．

1 乗算命令（MUL・IMUL）

乗算命令は，16ビットCPUを使用するうえで便利な命令です．8ビットCPUでは，一部の機種（6809など）を除いて乗算命令が存在しなかったため，たいへんわずらわしい計算をしなければならなかったのですが，8086では1命令での乗算が可能となり，とても使いやすくなりました．しかも符号付演算が可能ですから，プログラム作成上も容易になっています．

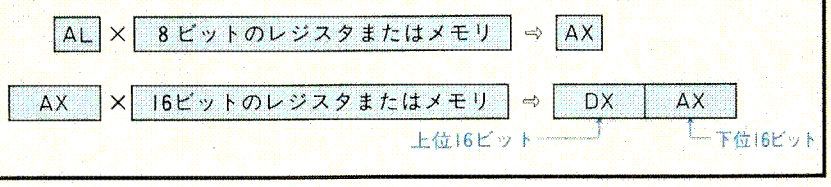
乗算命令には符号なし乗算命令（MUL：MULtiply accumulator by register/memory）と，符号つき乗算命令（IMUL：Integer MULtiply）とがあり，必要に応じて使いわけが可能です．しかもどちらも8ビット×8ビットあるいは，16ビット×16ビットのうち都合のよい方を使用することができます．

乗算命令は符号なし，符号付きとも可能

容易にわかることですが，8ビット×8ビットの乗算結果は16ビットに，16ビット×16ビットの乗算結果は32ビットに，と大幅にビット数が増えます．そのため，使用するレジスタに若干の制限があります．つまり乗算に使用するレジスタのうち少なくとも一方は，ALレジスタあるいはAXレジスタでなければならない，ということです．そしてALレジスタを使用した8ビット×8ビットの乗算の場合，演算結果の上位ビットがAHレジスタに，下位8ビットがALレジスタに格納されます．またAXレジスタを使用した16ビット×16ビットの乗算の場合，演算結果の上位16ビットがDXレジスタに，下位16ビットがAXレジスタ

に格納されます．これを図で表わすと図4.6のようになります．

図4.6



このような規則があるため，命令を書くときにはAL，またはAXレジスタに掛け合わせるもう一方のレジスタ，またはメモリのみを指定すればよいことになります．

命令としては次のようなものがあります．

乗算命令の整理

MUL　reg8 ……指定されたレジスタとALレジスタとの積をAXレジスタに格納する．

MUL　reg16……指定されたレジスタとAXレジスタとの積をDXレジスタとAXレジスタに格納する．

MUL　mem8　＊……指定されたメモリの内容とALレジスタとの積をAXレジスタに格納する．

MUL　mem16　＊……指定されたメモリの内容（逆ワード形式の2バイト）とAXレジスタとの積をDXレジスタとAXレジスタに格納する．

IMUL　reg8……指定されたレジスタとALレジスタの符号付数値との積をAXレジスタに格納する．

IMUL　reg16……指定されたレジスタとAXレジスタの符号付数値との積をDXレジスタとAXレジスタに格納する．

IMUL　mem8 ……指定されたメモリの内容とALレジスタの符号付数値との積をAXレジスタに格納する．

IMUL　mem16　＊……指定されたメモリの内容（逆ワード形式の2バイト）とAXレジスタの符号付数値との積をDXレジスタとAXレジスタに格納する．

※この命令群ではフラグの変化が複雑です．フラグの項を参照すること．

＊印が付いているのでわかると思いますが，メモリを指定する場合，BYTEまたはWORDの指定が必要です．ALレジスタが選択されるかAXレジスタが選択されるかは，もう一方のレジスタあるいはメモリのビット数で自動的に決定されるので注意してください．

例　題

符号付数値－1111HをAXレジスタに，同様に－1111HをBXレジスタに代入してその積を求め，得られた値の最上位バイトが110H，最下位バイトが113H番地に入るように答を格納せよ．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 B8EFEE        MOV     AX,0-1111
0007 BBEFEE        MOV     BX,0-1111
000A F7EB          IMUL    BX
000C 86E0          XCHG    AH,AL
000E 86F2          XCHG    DH,DL
0010 89161001      MOV     [0110],DX
0014 A31201        MOV     [0112],AX
0017 F4            HLT
```

16ビットの負の数を代入する場合，このように0から引いた式を入れればよい．

この場合にはMOV BX，AX としてもよい（その場合は機械語のアドレスがずれる）．

解　説

符号付数値をアセンブラから代入する場合，このように0から引く式を用いて入れることもできます．符号付数値の乗算ですから，IMUL命令を用います．計算結果はDXレジスタとAXレジスタに格納されますが，これをメモリに転送する際逆ワード形式で転送されますので，あらかじめXCHG命令により，上位バイトと下位バイトを入れ換えておきます．

負数の入力法

実行結果は次のとおりで，解答は正の数1234321Hであることがわかります．

```
hJG0,17↵
*1B00:0017
hJD110↵
0110 01 23 43 21 00 00 00 00 00 00 00 00 00 00 00 00          #C!
```

この場合BXレジスタを用いずに，IMUL AXとすることでAX＊AXを行なうこともできます．その他にAL＊AL，AL＊AH，AH＊DLといった組み合わせも可能ですが，実行後のもとのレジスタの値は変化してしまいます．

2 除算命令（DIV・IDIV）

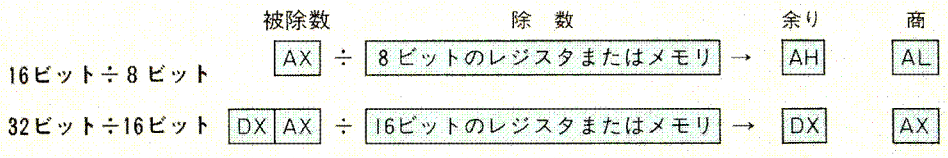
2-1　除算命令

16ビットCPUになって新しく設けられたもうひとつの命令に，除算命令があ

ります．除算機能を持つ8ビットCPUはありませんから，これも16ビットCPUの特長のひとつです．数学上では除算は乗算の逆演算ということになりますが，プログラム作成上では除算は乗算にはないいくつかの問題を持っています．

そのうちのひとつが，割り切れなかった場合の処理です．8086では扱う数値はすべて整数とみなしています．そこで割り切れなかった場合には，商と余りという形で結果を得ることになっています．具体的には16ビット÷8ビット→8ビットの商と8ビットの余り，または32ビット÷16ビット→16ビットの商と16ビットの余りという2つのうちいずれかを使用します．この場合も乗算命令同様使用するレジスタに制限があります．つまり16ビット÷8ビットの場合は，あらかじめ割られる数（被除数）をAXレジスタに，また32ビット÷16ビットの場合には，被除数の上位16ビットをDXレジスタに，下位16ビットをAXレジスタにセットしておかなければなりません．しかも，結果が格納されるレジスタの組み合わせも決まっています．この関係を図で示すと**図4．7**のようになります．

図4.7



2つめに注意しなければならないのは，符号の付いた除算の場合です．符号付き除算の場合には被除数と除数の組み合わせにより，4通りの符号の組み合わせが考えられます．しかも除算の場合には商と余り，という2つの結果が得られますから少し複雑です．次の例を考えてみてください．

符号の付いた除算

－8を3で割る場合，

商＝－2，余り＝－2

とする場合と，

商＝－3，余り＝1

とする場合とが考えられます．

除算の符号は，被除数＝余り

どちらも数学的には正しいのですが，ひとつの演算に対して2通りの答が出てくるのでは困りますから，どちらか一方に統一しなければなりません．**8086では，被除数の符号と余りの符号が一致するように決められています**（注：次ページ参照）．

表4.1

被除数	除　数	商	余　り
正	正	正	正
正	負	負	正
負	正	負	負
負	負	正	負

これを表にまとめると，**表4．1**のようになります．

あえて説明する必要はないと思いますが，商の符号は乗算の場合と同様，被除数と除数の符号が一致したときに正，一致しないときに負となり，一意的に定まります．

注）除数は減算の繰り返しとして計算される．わかりやすいように10進数で考えてみると，次のようになる．

15÷4の場合　　15から4を，1回ひくと　15－4＝11
2回ひくと　11－4＝7
3回ひくと　7－4＝3
4回ひくと　3－4＝－1　←付号が変わる
したがって答は3

つまり被除数から除数をどんどんひいていって，**余りの符号が反転したら，その直前の商と余りを答として採用する**わけである．はじめは被除数と余りの符号は一致しているので，最後の答でも被除数と余りの符号が一致するようになるわけである．この例を負の場合についても考えてみよ．なお，これとは異なった考え方をするCPUもある．

2-2　0による除算の処理

さて除算を行なう場合に，もうひとつ忘れてはならない重要なことがあります．エッ？何だっけ，なんていう人は，ここまで読み続けて少し疲れている証拠ですから，一服してからもう一度考えてみてください．……そう，除算を行なう場合は0で割ってはいけないという規則がありましたね，これのことです．

しかし1回だけ計算を行なう場合ならともかく，複雑な計算の途中で除算を使用するときには除数に0がこないとは限りません．そんな場合にはどうしたらよいでしょうか．

除算命令を実行する前に除数をチェックし，0なら特別な処理を行なうプログラムに分岐するのが一方法です（比較・分岐命令については，あとで説明します）．この方法は，最も確実で有効な手段ではありますが，すべての除算命令の前にいちいちチェックをしなければならない，という点でわずらわしさは否定できません．

そこで8086では除算命令自体の中に，除算命令で0で割る場合が現われたら，特別なプログラムに分岐するという機能が備わっています．具体的にいうと，0で割る除算が現われたら，INT 0命令を自動的に発生して除算命令を中断し，エラー処理ルーチンへ分岐するようになっています．INT 0命令は割り込み命令と呼ばれ，詳しくはあとで説明しますが，N88BASICにおけるON ERROR GOSUB文と似たような機能を果たす，と考えてもほぼ同じです．つまりエラーの起きたときだけ，指定された処理を行なう命令です．

0で割るとINT0が発生

8086はINT 0命令を実行すると，割り込みベクタと呼ばれる部分に書かれているアドレスを参照して分岐し，処理をそちらに移します．そのあとの処理命令はROMまたはRAM上に書かれていますので，処理の方法はマイコンの設計者やユーザが自由に定めることができます．詳しい動作については，後述の「割り込み」の項で説明しますが，PC-9801のN88-BASIC(86)では，通常，何の処理もせずに元のプログラムにもどるように書かれています．したがって，0で割る命令が出てきた場合には除算命令を実行せずに次のプログラムに移ります．

繰り返しますが，これはあくまでN88-BASIC(86)での話であって，機種やOS

が異なれば処理方法も変わってきます（たとえば，警告を発生して処理を中断するなど）．したがって8086自体は0で割る命令を実行すると，INT0命令を発生してそちらに処理を移すだけです．

少し長くなりましたが，命令のまとめと問題に入る前に，除算命令を使用する際に制限となる事項について述べておきます．

2-3　除算時のオーバーフロー

16ビット÷8ビットの場合について考えてみましょう．この除算は，前述したとおり，

被除数	除数		商	余り
16ビット	÷ 8ビット	⟶	8ビット	8ビット

という形式で行なわれます．この形式には実は不都合な点があります．お気付きになりましたか．そうです，この形式では，商が8ビットに納まらない場合に困ってしまうわけです．具体例を考えてみるとわかりやすいので，例をあげましょう．たとえば2222Hを2Hで割るとどうなるでしょうか．

結果がオーバーフローするときも，タイプ0割り込みが発生

2222H ÷ 2H → 1111H　余り0

となれば正しい答なのですが，16ビットの除算命令では商は8ビットに限られていますから，正しい答を得ることができません．8086では，この場合も0で割った場合と同様にINT0命令を発生し，除算を中断します．実際には被除数の上位8ビットと除数の8ビットを比べて，被除数の上位8ビットの方が大きい場合には商が8ビットに納まらないことがわかりますから，その時点でINT0命令を発生します．符号付き，符号なしにかかわらず，同様のチェックが行なわれます．この制限を越える数値については別にプログラムを作る必要があり，このことが除算命令を使用するうえで不便な点です．

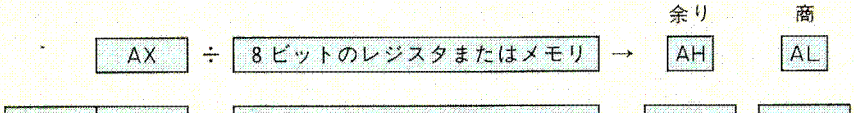
しかし実際には，16ビット÷8ビットといった除算をすることはあまりなく，8ビット÷8ビットのように被除数と除数のビット数が同じである場合がほとんどですから，この場合には制限を越える心配はありません．なお8ビット÷8ビットのような除算は符号拡張命令などを用いて16ビット÷8ビットに変換して行ないます．以上で述べてきたことは，32ビット÷16ビットの除算についてもまったく同様に考えられます．

2-4　除算命令の使用法

それでは除算命令の使用法に入りましょう．除算命令には，符号なし除算命令（DIV：DIVision）と符号付き除算命令（IDIV：Integer DIVision）とがあります．被除数はあらかじめ特定のレジスタに入れておかなければなりません．また，結果の商と余りも特定のレジスタに返されます．除数としては8ビットまたは16ビットのレジスタまたはメモリを使用することができます．除数として与える数値のビット数によって，16ビット÷8ビットの除算か，32ビット÷16ビットの除算かが決定されます．具体的には次ページ図4.8のようになります．

被除数はAX, DX
商はAH, DH
余りはAL, AX

図4.8



除算命令の整理

DIV　reg 8 ……AXレジスタの内容を指定したレジスタの内容で割り，商をALレジスタに，余りをAHレジスタに格納する．

DIV　reg16……DXレジスタとAXレジスタの組で表わされる内容を指定したレジスタの内容で割り，商をAXレジスタに，余りをDXレジスタに格納する．

DIV　mem 8　*……AXレジスタの内容を指定したメモリの内容で割り，商をALレジスタに，余りをAHレジスタに格納する．

DIV　mem16　*……DXレジスタとAXレジスタの組で表わされる内容を指定したメモリ（逆ワード形式の2バイト）で割り，商をAXレジスタに，余りをDXレジスタに格納する．

IDIV　reg 8 ……AXレジスタの内容(符号付数値）を指定したレジスタの内容（符号付数値）で割り，商をALレジスタに，余りをAHレジスタに格納する．

IDIV　reg16……DXレジスタとAXレジスタの組で表わされる内容（符号付数値）を指定したレジスタの内容（符号付数値）で割り，商をAXレジスタに，余りをDXレジスタに格納する．

IDIV　mem 8　*……AXレジスタの内容（符号付数値）を指定したメモリの内容（符号付数値）で割り，商をALレジスタに，余りをAHレジスタに格納する．

IDIV　mem16　*……DXレジスタとAXレジスタの組で表わされる内容（符号付数値）を指定したメモリの内容(逆ワード形式の2バイトの符号付数値）で割り，商をAXレジスタに，余りをDXレジスタに格納する．

※この命令群を実行後，フラグの値は不定になる．

注1）除算の結果が格納すべきレジスタに納まらない場合には，タイプ0の割り込み(INT0)が発生し，除算は中断される．この割り込み処理が終了すると，除算命令の次の命令から実行を開始する．

注2）符号付数値の除算命令（IDIV）では，結果が8ビットの場合には－127d～＋127dの範囲に，また16ビットの場合には，結果が－32767d～＋32767dの範囲に納まらなければならない．すなわち，結果が8ビットでは－128d(＝80H)，16ビットでは－32768d（＝8000H）となる除算は行なうことができない．この場合には上記と同様にINT 0が発生する．なお，インテル社の和文マニュアルには，これの記載がないので注意を要する．

〔例〕－128d÷1d（16進で表わすと0FF80H÷1H）を考える．結果として－128d（＝80H）が得られるはずである（格納先のレジスタに納まる）が，実際にはINT0が発生し，結果は求まらない．

例　題

符号なし16ビット数値1234Hを100H番地に格納し，これを符号なし8ビット数値56Hで割った商を110H番地に，余りを111H番地に格納せよ．また同じ数値を10Hで割った場合にはどうなるか．除算命令実行後のAXレジスタの値を120H番地に格納せよ．また同様に0Hで割った場合のAXレジスタの値を130H番地に格納せよ．

解　答

```
0000 8CC8            MOV     AX,CS
0002 8ED8            MOV     DS,AX
0004 C70600013412    MOV     WORD [0100],1234
000A A10001          MOV     AX,[0100]
000D B356            MOV     BL,56
000F F6F3            DIV     BL
0011 A31001          MOV     [0110],AX
0014 A10001          MOV     AX,[0100]
0017 B310            MOV     BL,10
0019 F6F3            DIV     BL
001B A32001          MOV     [0120],AX
001E A10001          MOV     AX,[0100]
0021 B300            MOV     BL,00
0023 F6F3            DIV     BL
0025 A33001          MOV     [0130],AX
0028 F4              HLT
```

直接数値で割り算をする命令はないので，あらかじめレジスタに値を代入する．

BLレジスタを指定した除算なので，被除数はAXレジスタにあるものとみなされる．

この方法で指定通りに格納される．

解　説

あらかじめAXレジスタに被除数を格納しておきます．除数はBLレジスタでなくても，8ビットとして使用できるレジスタなら何でもかまいません．除算の結果，AXレジスタの上位8ビットであるAHレジスタには余りが，下位8ビットであるALレジスタには商が格納されているので，指定どおりの方法でメモリに格納するためには，わざわざAH，ALの2回に分けて転送する必要はなくAXレジスタとして一度に転送すればよいわけです．ここで結果をみてみます．

```
hJG0,28
*1B00:0028
hJD100,13F
0100  34 12 00 00 00 00 00 00 00 00 00 00 00 00 00 00        4
0110  36 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00        6
0120  34 12 00 00 00 00 00 00 00 00 00 00 00 00 00 00        4
0130  34 12 00 00 00 00 00 00 00 00 00 00 00 00 00 00        4
```

1234H÷56Hの結果は商が36H，余りが10Hであることがわかります．

ところで1234H÷10Hの結果はというと，本当は商が123H，余りが4Hのはずですが，商が8ビットを越えてしまうため正しく求められていません．AXレジスタの内容としては1234Hが残っています．また1234H÷0Hの場合も答は求められずに，AXレジスタの値として1234Hが残っています．しかしこれでは，正しく結果が求められたか否かはわかりません．このためにもINT 0割り込みが生じた場合の処理法を定めておかなければならないのですが，現段階ではまだ程度を越えているので説明は省きます．全編を読み終えて8086をひととおり理解したあとで試みられるとよいでしょう．

なお8086の文法書では，除算の結果フラグは不定になり（フラグについては後述），またINT0割り込みが生じた場合AXレジスタの内容は不定になるとされていますが，上記の結果からわかるとおり，もとのAXレジスタの内容が保存されるものと思われます．16ビット÷8ビットについては，筆者の使用したシステムでは，除算のできない場合にはAXレジスタが保存されることが確認されていますが，AXレジスタを用いて除算が行なわれたか否かを判別することはできません．

例）01FFH÷02H——→商0FFH，余り01Hで，AXレジスタは01FFHのまま．

練習問題

4—6

バイト数値11Hを100H番地に，バイト数値0FFHを101H番地に代入し，これを符号なし数値とみた場合の積を110H，111H番地に格納せよ．また，これを符号付数値とみた場合の積を112H，113H番地に格納せよ．番地指定にはBXレジスタによる間接指定法を用い，結果は逆ワード形式でもよい．

解答

```
0000 8CC8        MOV     AX,CS
0002 8ED8        MOV     DS,AX
0004 BB0001      MOV     BX,0100
0007 C60711      MOV     BYTE [BX],11
000A C64701FF    MOV     BYTE 01[BX],FF
000E 8A07        MOV     AL,[BX]
0010 F66701      MUL     BYTE 01[BX]
0013 894710      MOV     10[BX],AX
0016 8A07        MOV     AL,[BX]
0018 F66F01      IMUL    BYTE 01[BX]
001B 894712      MOV     12[BX],AX
001E F4          HLT
```

ALレジスタに一方のデータを転送しておく．

メモリとの乗算なのでTYPEの指定が必要である．

前の乗算の結果，ALレジスタの内容は変化しているので，再び転送する必要がある．

解　説

一方がメモリの場合は，BYTEまたはWORDの指定が必要であることに注意してください．一度乗算を行なうと，結果に従ってAL，AHレジスタは書き換えられますから，二度めの乗算を行なう際にはもう一度値を転送し直さなければなりません．結果は次のとおりで，符号なしの場合は10EFH，符号付きの場合は0FFEFH(−11H)，のようにまったく違う結果となります．

```
hJG0,1E
*1B00:001E
hJD110
0110 EF 10 EF FF 00 00 00 00 00 00 00 00 00 00 00 00
```

4−7

符号付16ビット数値−1234Hを符号付16ビット数値−567Hで割り，商を110H番地以後に，余りを120H番地以後に格納せよ．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 B8CCED        MOV     AX,EDCC
0007 99            CWD
0008 BB99FA        MOV     BX,FA99
000B F7FB          IDIV    BX
000D A31001        MOV     [0110],AX
0010 89162001      MOV     [0120],DX
0014 F4            HLT
```

このように入力すると，後でLコマンドで確認した時には正しい形になっている．

符号拡張命令

解　説

同じビット数どうしの除算を行なう場合には，符号拡張命令を用いてあらかじめ被除数のビット数を増やしておく必要があります．

なお符号なし数値で同様の場合には，符号拡張命令は使用せず，DXレジスタまたはAHレジスタに直接数値0を代入します．

符号付数値か符号なし数値かは，あくまでもプログラムを作る人間が決めることであって，機械はその数値が符号付きか符号なしかを判断することはできないのだ，ということを忘れないでください．実行してみると，次のようになり，余りが0FE01で，負の数であることがわかります．

```
hJG0,14
*1B00:0014
hJD110,12F
0110 03 00 EF FF 00 00 00 00 00 00 00 00 00 00 00 00
0120 01 FE 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

5 ランダムテクニックとフラグレジスタの働き

1 ランダムテクニック

いままで作成してきたプログラムはすべて最後がHLT命令で終わり，実行する場合はG0，23↵のようにHLT命令のある番地をブレークポイントに指定しなければなりませんでした．ブレークポイントの指定を誤ったり忘れたりすると，正しく実行されなかったりホールトしてしまい，リセットをかけなければならなくなるなど面倒です．そこで次のような方法を考えてみましょう．

機械語の90はノーオペレーション

いままでプログラムを作る場合，まず，

```
h]F0, 1FF, 0↵
```

のように，プログラム領域を0でクリアしていました．これを，

```
h]F0, FF, 90,
```

として，プログラム領域を90Hで埋めます．Lコマンドで見てみるとわかるように90HはひとつでNOP(ノーオペレーション)，すなわち何も実行しないことを表わす命令です．こうしておいて，短いプログラムを作ったあと，

```
h]G0, FF↵
```

としてください．ちゃんと終わったでしょう．こうすればブレークポイントをいちいち考えずにG命令を使うことができて便利です．

INT 1EはBASICにもどる命令

ついでに，プログラムの最後にAコマンドで，

```
h]AFE↵
00FE  CD1E      INT     1E
```

と入力したうえで，

```
h]G0↵
```

としてみてください．画面がクリアされて“OK”の表示が出たはずです．こ

れはマシン語からベーシックにもどる命令なので，再びMONとしてモニタモードでメモリを確認してみてください，正しく実行されているはずです．

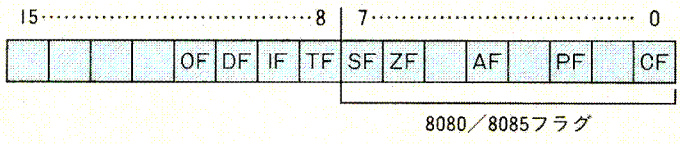
なおこの方法ではレジスタの値が変化してしまうので，レジスタの値をあとで見たいときには使用できませんが，マシン語プログラムの終了法としては最も簡単で便利な方法なので，紹介しておきました．

2 フラグレジスタの働き

演算の説明の中でもフラグレジスタの働きについて少し触れてきましたが，この章の結びとして，フラグレジスタの各々のフラグの働きについて説明しておきましょう．なおフラグの中にはまだ学習していない働きをもつものもありますが，それらについてもすべて触れておきますので，本書をひと通り読み終わった後，本項にもどって読み返していただくと一層理解が深まると思います．

フラグは，演算命令やフラグ制御命令の結果に従って各ビットが単独でセットあるいはリセットされ，フラグを参照する命令に影響を与えたり，直接CPUの動作に影響を与えるという作用を持っています．以下に各ビットの働きを述べます．

図4.9



ビット7からビット0までのフラグは，8086の前身である8ビットCPU8080および8085のフラグと同じ構成と機能を持っています．ビット15からビット8までのフラグは8086になって新しく加えられたフラグです．

以下の6つのフラグは，演算の結果のCPUの状態を示すフラグで，**ステータスフラグ（状態フラグ）**と呼ばれています．

CF●キャリー・フラグ

8または16ビットの演算の結果，最上位ビットからの桁上がり（キャリー），または借り（ボロー）が生じた場合に1にセットされ，これらが生じなければ0にリセットされます．また論理演算を行なった場合も0にリセットされます．INC命令，DEC命令では桁上がりや借りが生じてもCFの内容は変化しません．シフト，ローテート命令などを実行すると，最上位ビットあるいは最下位ビッ

トの隣りにあるビットとして扱われ，ビットの内容が転送されます．

PF●パリティ・フラグ

8または16ビットの演算の結果のレジスタまたはメモリの内容で，1となったビットの数が偶数個の場合に1にセットされ，奇数個の場合は0にリセットされます．これは，主としてデータ転送の際にビットエラーが発生したか否かを調べるために使用されます．

AF●補助キャリー・フラグ

8ビットの演算を行なった際に，下位4ビットから上位4ビットへの桁上がりあるいは借りが生じた場合に1にセットされ，生じなかった場合は0にリセットされます．また16ビットの演算を行なった際には，下位8ビットから上位8ビットへの桁上がりあるいは借りが生じた場合に1にセットされ，生じなかった場合は0にリセットされます．このフラグは10進演算を行なった際の10進補正命令において使用されます．

ZF●ゼロ・フラグ

8ビットまたは16ビットの演算の結果，レジスタまたはメモリの内容がゼロとなったときに1にセットされ，それ以外の場合0にリセットされます．

SF●サイン・フラグ

8または16ビットの演算の結果，最上位ビットが1の場合に1にセットされ，0の場合は0にリセットされます．つまり，演算の結果を符号付数値とみた場合，負のときは1にセットされ，正あるいは0のとき0にリセットされることになります．

OF●オーバーフロー・フラグ

8または16ビットの数値を符号付数値とみなして演算した際に，オーバーフローあるいはアンダーフローが生じた場合に1にセットされ，結果が範囲内に納まった場合には0にリセットされます．

たとえば8ビットの数値40Hと50Hを加算した場合，結果は90Hとなります．これは，符号なし数値とみた場合には正しい結果ですが，符号付数値とみた場合はオーバーフローして，正しい結果になっていません（8ビット符号付数値の最大値は7FHであり，90Hは負の数とみなされる）．このような場合にOFがセットされます．扱う数値が符号付数値であるか否かは，プログラマーの決定することであり，CPUにはその判断力はないので，どのような場合でも符号付数値の演算を行なったと仮定して，フラグがセットあるいはリセットされま

す.

動作原理は，〈最上位ビットからCFへのキャリーが発生した〉か，または，〈2番めのビットから最上位ビットへのキャリーが発生した〉かのどちらか一方のみが起こった場合にセットされます（理由は自分で考えてください）.

以下の3つのフラグは，プログラムによってセットまたはリセットし，CPUの動作をコントロールするフラグであり，**コントロール・フラグ（制御フラグ）**と呼ばれています.

コントロール・フラグ

DF●ディレクション・フラグ

ストリング操作を行なう際に，ソースおよびディスティネーションアドレスを自動的に増加させるか，減少させるかの方向を決めます．CLD（CLear Direction flag）命令で0にクリアすると，アドレスは増加する方向に向きます．STD（SeT Direction flag）命令により1にセットされ，アドレスは減少する方向に向きます(**「ストリング命令」**参照).

IF●インタラプト・イネーブル・フラグ

INTR端子への，外部からの割り込み（インタラプト）を受け付けるか否かを決定します．STI（SeT Interrupt-enable flag）命令により1にセットされ，INTR端子から入力される外部割り込み要求を受け付けることが可能となります.

CLI（CLear Interrupt-enable flag）命令により0にリセットされ，割り込み禁止状態になります（割り込み要求が無視される）．NMI端子から入力される外部割り込み要求（ノンマスカブルインタラプト）および，プログラム上に書かれた内部割り込み（プログラム割り込み）は禁止できません．内部割り込みが発生した際は，その処理が終わるまでの間，自動的に0にクリアされます(**「割り込み」**の項参照).

TF●トラップ・フラグ

TFが1にセットされるとCPUはシングルステップモードになり，一命令実行ごとに自動的に内部割り込み（INT 1）を発生し，割り込み処理ルーチンに分岐します．このモードはプログラムをデバッグするのに用いられ，一命令実行ごとにその結果の確認ができます．TFが0にリセットされると，CPUは通常どおり連続して命令の処理を行ないます.

なお，内部割り込みが発生すると，その処理が終わるまでの間自動的に0にクリアされますが，処理が終了して元のプログラムに復帰すれば，フラグは元の状態に復帰されます．このフラグをセット，リセットするためには，スタッ

クを介して，レジスタ上で直接操作を行なわなければなりません（**「論理演算」**の項に具体例が掲げてある）．

では次に，フラグを制御する命令について整理しておきます．

フラグ制御命令の整理

STC(SeT Carry flag)……キャリー・フラグを1にセットする．
CLC(CLear Carry flag)……キャリー・フラグを0にリセットする．
CMC(CoMplement Carry flag)……キャリー・フラグが0であれば1に，1であれば0にセットし直す．
STD(SeT Direction flag)……ディレクション・フラグを1にセットする（アドレスを自動的に減少させる方向）．
CLD(CLear Direction flag)……ディレクション・フラグを0にリセットする（アドレスを自動的に増加させる方向）．
STI(SeT Interrupt-enable flag)……インタラプト・フラグを1にセットし，割り込み受け付け可能状態にする．
CLI(CLear Interrupt-enable flag)……インタラプト・フラグを0にセットし，割り込み禁止状態にする．

フラグレジスタ関係の命令はこの他に，フラグレジスタの下位8ビットとAHレジスタの間での転送命令および，フラグレジスタをスタックにPUSH，POPする命令があります．なお，PUSH，POPの詳しい動作については**「スタック」**の項で説明します．

フラグ転送命令の整理

LAHF(Load AH from Flags)……フラグレジスタの下位8ビットをAHレジスタに転送する．未使用ビットの値は不定．
SAHF(Store AH into Flags)……AHレジスタの内容をフラグレジスタの下位8ビットに転送する．
PUSHF(PUSH Flags onto stack)……フラグレジスタの内容をスタック上にPUSHする．
POPF(POP Flags off stack)……スタック上のワードデータをフラグレジスタにPOPする．

第5章

繰り返し命令と入出力

1　比較・分岐命令

2　BASICで機
　　械語を

3　サブルーチンとス
　　タック

11 比較・分岐命令

1 繰り返しの必要なプログラム

ここまでの練習で，皆さんは演算命令が使いこなせるようになったと思います．しかし実際にコンピュータに仕事をさせるためには，これだけの命令では十分ではありません．何が必要でしょうか？そう，繰り返し計算です．わかりやすい例で考えていきましょう．

1-1 FOR～NEXT

和を求める計算

1から100までの数（10進数）の和を求める場合について考えてみましょう．このプログラムをBASICで作るとすれば，次のようになります．

```
100 SUM=0
110 FOR I=1 TO 100
120 SUM=SUM+I
130 NEXT I
140 PRINT SUM
```

ここではFOR～NEXTのループ中で，Iを1から100まで1ずつ増しながらSUMに加算していきます．ここでの考え方は，Iを1ずつ増やして100になるまで繰り返す，という考え方です．

1-2 IF～THEN

アセンブルするにはFOR～NEXTでなくIF～THEN

アセンブリ言語でもこれと同じことをやれば，1から100までの和を求めることができるわけです．しかし，これをそのままアセンブリ言語に直すわけにはいきません．それは，アセンブリ言語にはFOR～NEXT文のように2つ1組で使用する命令がないからです．そこでFOR～NEXT文を使用せずにIF文を使用して，同じプログラムを作ってみます．

```
100 SUM=0
110 I=0
120 I=I+1
130 SUM=SUM+I
140 IF I<>100 THEN 120
150 PRINT SUM
```

ここではIを1ずつ増加させて加えていき，Iが100でなければ加算を繰り返し，Iが100になれば終わりにするようにしています．このように，ある変数の値（この場合はI）を別の数と比べることによって分岐する命令（140行）のことを条件分岐命令といいます．変数の値にかかわらず分岐する命令（GOTO文のような命令）のことを無条件分岐命令といいます．

減少法のメリットは0で判断

このプログラムのままでも、アセンブリ言語に直すことができるのですが，もう少し簡単にするために次のように変形します．今度はIを100から1ずつ減らして1まで加算していきます．

```
100 SUM=0
110 I=100
120 SUM=SUM+I
130 I=I-1
140 IF I<>0 THEN 120
150 PRINT SUM
```

なぜわざわざ順序を逆にしたのでしょうか？この方法のメリットの1つは，加える数の上限値（この例では100）にかかわらず，終了条件を0になった場合とすることができる点にあります．

さてそれでは，このプログラムに従ってアセンブリプログラムの方法を考えていきましょう．

例　題

1から100までの数（10進数）の和を求め，結果を100H，101H番地に格納しなさい．結果は逆ワード形式の16進数でよいものとする．

解　答

```
0000 8CC8      MOV     AX,CS
0002 8ED8      MOV     DS,AX
0004 B96400    MOV     CX,0064
0007 B80000    MOV     AX,0000
000A 01C8      ADD     AX,CX
000C 49        DEC     CX
000D 75FB      JNE     000A
000F A30001    MOV     [0100],AX
0012 F4        HLT
```

10進数の100(以後100dのように書く)は，16進数で64H ですから，これをCXレジスタに代入します．

AXレジスタに加算していきます．

CXの内容を1減じる命令です．

CXの内容が0でなければ，00A番地へ分岐する命令です．

解　説

前掲のBASICプログラムを基本にして，アセンブリ言語で書いてみます．合計を格納するために変数SUMではなくAXレジスタを，100から1まで変化する変数としてIのかわりに，CXレジスタを用いています．初期値としてAXレジスタに0，CXレジスタに100（16進数で64H）を代入した後，AXレジスタにCXレジスタの値を加えています．

次に，CXレジスタの内容を1減じる必要がありますが，そのための命令として，ここではDEC命令（DECremnt）を用いています．この命令は，オペランドで示すレジスタあるいはメモリの内容を1だけ減じる命令です．

DECrement

```
DEC    CX
```

1減らすDEC命令

は，BASICで言えば，

```
I=I-1
```

に相当します．

さて，問題はその次です．次の命令は前の命令，すなわちDEC　CXの結果が0でなければ，オペランドに示す番地にジャンプせよ，という命令です．JNEとは，Jump if Not Equalという意味であり，この場合のEqualとは〝＝0〟という意味です．この命令とまったく同じ命令を，JNZ（Jump if NoT Zero）というニーモニックで書くこともできます．これも前述の命令同様，前の命令の結果が0でなければ分岐せよ，という命令です．

この2つの命令は，実はまったく同じものであり，どちらを書いても生成する機械語は同じです．これは，プログラムを組む人間にとって便利なように，どちらも使えるようになっているだけなのです．

Equal＝Zero

```
DEC    CX
JNZ    000A
```

は，

```
I=I-1
1F  I<>0  THEN  120
```

という部分に相当します．

このように条件付き分岐命令（ここではJNZ）は，**直前の演算結果に基づいて，分岐するか否かを決定します．**重要なことはJNZの場合その結果の判断が，直前の演算結果が0か否か，で行なわれるという点です．わざわざBASICプログラムの変数の変化する順序を逆にしたのは，このためです．

条件付き分岐命令

さて，ここで処理のアルゴリズムをフローチャートにしてみます．今までのプログラムは流れが一本だったので，特にフローチャートにする必要はなかったのですが，このプログラムのように分岐命令を含んだものは，フローチャートにして確認することが必要です．

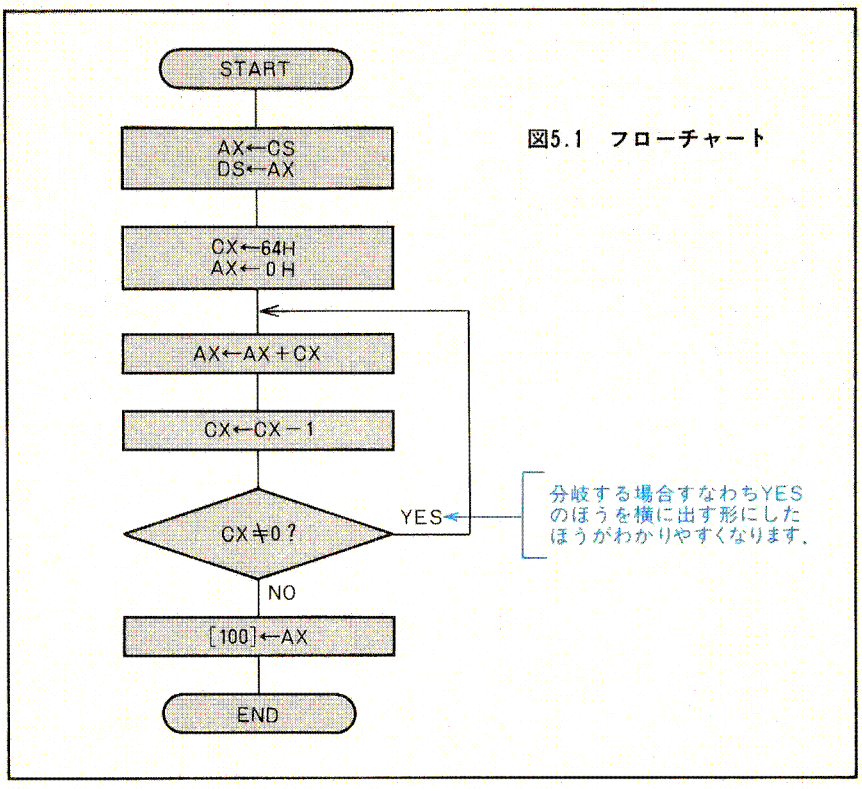


図5.1　フローチャート

結果を確認してみましょう．

```
hJG0,12↵
*1B00:0012
hJD100↵
0100 BA 13 00 00 00 00 00 00 00 00 00 00 00 00 00 00        ｺ
```

合計は13BAH（＝5050d）となっています。

まだしっくりこないナ，という人のためにもうひとつ例題をやってみましょう．

例　題

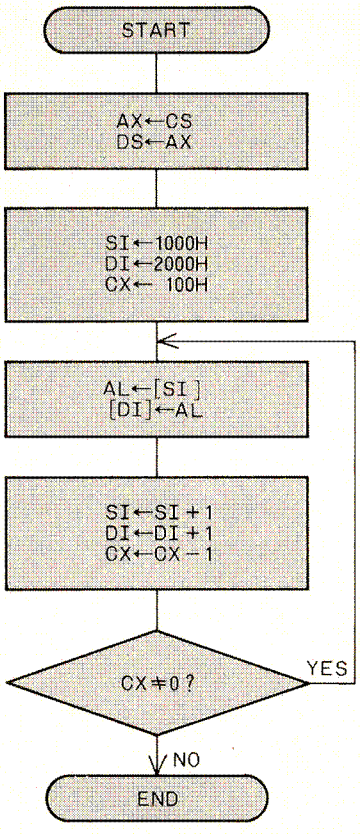
1000H番地以後に100H個のバイトデータがある．これを2000番地からさらに転送しなさい．

考え方

SIレジスタに1000Hを，DIレジスタに2000Hを代入し，1個転送するごとにこれらの値を1つ増加させる（INC命令を使う）．またデータの個数をCXレジスタに代入し，1個転送するごとに値を1つ減じ，0になったら終了する．

わかりやすくするために，処理のアルゴリズムをフローチャートにしてみましょう．

図5.2
フローチャート



解　答

```
h]L0,16
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 BE0010        MOV     SI,1000
0007 BF0020        MOV     DI,2000
000A B90001        MOV     CX,0100
000D 8A04          MOV     AL,[SI]
000F 8805          MOV     [DI],AL
0011 46            INC     SI
0012 47            INC     DI
0013 49            DEC     CX
0014 75F7          JNE     000D
0016 F4            HLT
```

SI, DI：指示したレジスタ（またはメモリ）の内容に，1加える命令です．

000D：分岐先のアドレスをオペランドとして指定します．

解　説

どうですか？フローチャートを書いてからプログラムを作成すると，作成が容易で誤りが少なくなりますね．プログラムを組む場合は，簡単なプログラムであっても，フローチャートを作成する習慣をつけておくと，あとで修正を加える必要が生じたときなども，きわめて楽になります．

このプログラムでは，1個のデータを転送した後，アドレスを指定するためのレジスタであるSI，DIの各レジスタの値を1つ増やします．次に転送したデータの個数をカウントするためにCXレジスタの値を1つ減じ，その結果が0でなければ繰り返し次のデータを転送します．BASICでも同じですが，転送個数が本当に正しいのかどうかをよく確かめる必要があります（この場合なら，100H個送るべきところを101H個，あるいは0FFH個送るようになっていないかどうか）．この場合は正しいですね．

INC命令（INCrement）は，指定したレジスタあるいはメモリの内容に1を加える命令です．先のプログラムもそうでしたが，回数や個数を数えるレジスタとしては，CXレジスタ（カウンタ・レジスタ）を用いるのが一般的です．それは，このレジスタをカウンタとして用いた場合にのみ有効な命令（ストリング命令やLOOP命令など）があるためですが，詳しくは後章で説明します．

1増やすINT命令

カウントはCXレジスタで

では，実行してみましょう．とはいっても，このままでは正しくできたのかどうかよくわからないので，

h] F1000, 21FF, 0↵

として，メモリをクリアしたうえで，

メモリのクリア

h] F1000, 11FF, FF↵

として，1000H番地から11FFH番地までに0FFHを代入しておきます．

```
hJG0,16↵
*1B00:0016
hJD2000,210F↵
2000 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
2010 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
2020 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF

20E0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
20F0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
2100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
hJ
```

転送された個数も，正しく100H個となっているのがわかりますね．

2 条件分岐命令とフラグレジスタ

ところで今まで，条件分岐命令は前の命令の演算結果に従って分岐するか否かを決める，と言ってきました．たとえばJNZ命令の直前にDEC　CX命令がある場合はCXレジスタの値を1減じ，その結果が0でなければ分岐する，ということでした．しかしそれでは，条件分岐命令は必ず演算命令と組み合わせなければならないのでしょうか？実は，そんなことはありません．あとで述べる比較命令や，論理演算命令の次にもってくることもできますし，それらの命令

のしばらく後に持ってくることも可能です．

条件分岐命令はフラグで分岐

結果から言いますと，**条件分岐命令は，正確には，前の演算をみているのではなく，その結果としてセットあるいはリセットされたフラグレジスタの値によって分岐を決めているのです．**たとえばJNZ命令は，ゼロフラグの状態を調べ，ゼロフラグが0ならば（つまり直前の演算結果が0でないとき）分岐し，ゼロフラグが1ならば（つまり直前の演算結果が0のとき）分岐せずに次の命令を実行する命令なのです．普通は演算の結果によって，フラグがセットあるいはリセットされますから，分岐命令はフラグの値を調べて分岐することにより，「直前の演算結果に従って分岐する」ことができるわけです．

このことを利用して，比較命令や論理演算命令の結果に従って分岐させることができます．

では，次に比較命令を使用して分岐するプログラムを作ってみましょう．

例　題

前の例題で行なった1から100までを加えあわせる問題を，1から順に加えていき，100になったら終了する形で作りなさい．結果は100H，101H番地に格納すること．

考え方

今度は1から100まで数を増やしていきます．100になったかどうかを判別する命令としてCMP命令（CoMPare）を用います．

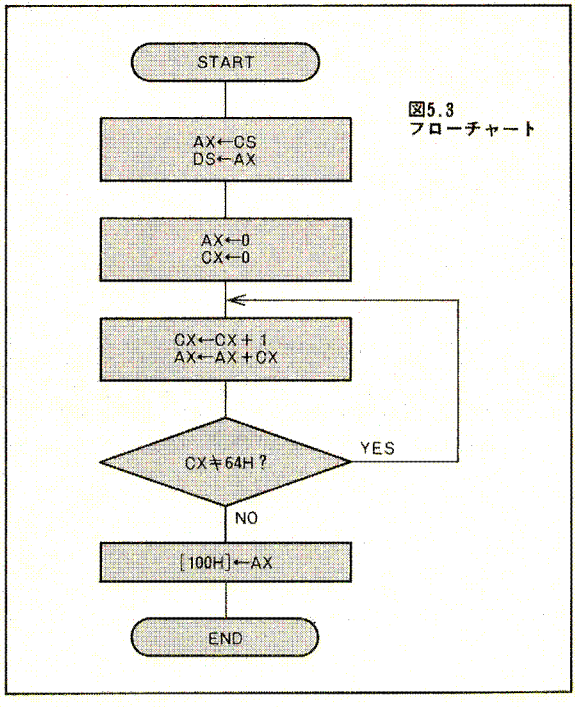


図5.3
フローチャート

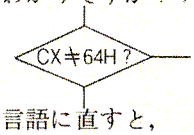
解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 B90000        MOV     CX,0000
0007 B80000        MOV     AX,0000
000A 41            INC     CX
000B 01C8          ADD     AX,CX
000D 83F964        CMP     CX,0064   ┐CXレジスタの値を，直接数値64H(＝100)と比較して
0010 75F8          JNE     000A      ┘います．
0012 A30001        MOV     [0100],AX
0015 F4            HLT
```

解　説

比較を行なうCMP命令

比較・分岐命令の使い方はおわかりですか？フローチャートでは，



で示された部分が，アセンブリ言語に直すと，

CMP　　CX, 0064
JNE　　000A

という2つの命令に分けられています．CMP命令により，CXレジスタの値と数値64Hとが比較されます．次にJNE命令（Jump if Not Equal）は，その結果がイコールでなければ000AH番地に分岐します．

ところで，JNE命令は前の演算結果が0でないときに分岐する，と前にいいましたね．ところがここでは，同じ命令が，前との比較の結果がイコールでないときに分岐する，という命令として使われています．これはどういうことかというと，前に述べたように，条件分岐命令はフラグに従って分岐しているからなのです．

ここで使用されているCMP命令は，実は減算命令と同じ働きをします．つまりこのCMP命令は，CXレジスタの内容から64Hを減じます．そこでもし，その結果が0であればゼロフラグがセットされます．〝減算結果が0である〟ということはつまり，2つの比較された数が〝イコールである〟こととまったく同等です．そこで前の問題では〝0になった場合〟と言い，この問題では〝イコールになった場合〟と言ったわけです．

CMP命令はもとのデータを変化させない

それでは，CMP命令と減算命令とはどこが違うのでしょうか？**CMP命令は，単に減算を行ない，結果に従ってフラグレジスタをセットするだけで，もとのレジスタやメモリの値を変化させない，**という点がこの2つの命令の相違点です．この例題では，CMP命令の前後でCXレジスタの値は変化せず，ただ比較の結果に従ってフラグレジスタが変化するだけです．もしここでSUB命令を使

用すると，CXレジスタにはもとのCXレジスタの値から64Hが減じられたものが格納されてしまい，つごうがよくありません．

以上のことから，演算結果が0であることと，比較結果がイコールであることとは同等であることがおわかりいただけたと思います．

3 分岐命令の種類

比較結果で分岐するその他の命令

ここまでの例題では，結果がイコールになる場合しか扱いませんでした．しかし，分岐する条件として，この他に〝より大〟，〝以下〟などといった条件判断を使う場合もでてきます．これらの命令を簡単に整理しておきましょう．

なお，条件の表わし方として，たとえば，

A≧B

には，AはB以上であるという表わし方と，AはBより小さくないという表わし方の2通りの表現法があります．PC-9801の内蔵アセンブラでは，このどちらの表わし方も使用できることになっています．しかしどちらの命令もまったく同じものでですから，生成される機械語も同じものになります．つまりA≧Bなら分岐する，という命令には，

大小比較の表記法

JAE(Jump if Above or Equal)：大きいかまたは等しい（＝以上）
JNB(Jump if Not Below)　　：小さくない

の2通りの表現があるわけです．なお，機械語は同じですから，逆アセンブルした場合は，どちらか一方の表現法に統一されます（後載の表参照）．

用語の使い方について注意しておきます．〝より大きい〟，〝より小さい〟という表現はその数は含まないという意味，つまり〝＞〟，〝＜〟に相当し，〝以上〟〝以下〟はその数を含むという意味，つまり〝≧〟，〝≦〟に相当します．〝以上〟，〝以下〟は，それぞれ〝大きいかまたは等しい〟，〝小さいかまたは等しい〟と表現されることもあります．

符号なし数値と符号付き数値で命令が異なる

大小比較の際に忘れてはならないことが，もうひとつあります．それは，扱う数値が符号付き数値か符号なし数値かによって結果が違ってくる場合がある，ということです．たとえば，01Hと0FFHを比較したとしてみましょう．符号なし数値と考えれば0FFH＝255dですから，0FFHの方が大きいことになります．ところがこれを符号付き数値と考えると0FFH＝－1dですから，01Hの方が大きいことになります．扱う数値を符号付きとみた場合と符号なしとみた場合とで結果が違ってきますから，条件分岐命令にも2通り用意されています．

具体的には，

JAE (Jump if Above or Equal)

の場合，符号なし数値の比較であり，

JGE(Jump if Greater or Equal)

の場合，符号付き数値の比較になります．

符号付き数値の比較

大小を表わす言葉には，次のように2系列あります．

	〔大〕	〔小〕	〔等〕
符号なし数値の場合	Above	Below	Equal
符号付き数値の場合	Greater	Less	Equal

なお，等しいか否かのみを比較する場合には，〝Equal〟のかわりに〝Zero〟を使用することもできます．

これらの関係を，簡単に表にしておきましょう．

比較命令表

分岐するための A，Bの大小関係*	符号無し数値の場合	符号付き数値の場合
A ＞B	JA ／JNBE	JG ／JNLE
A ≧B	JAE／JNB	JGE／JNL
A ＝B	JE ／JZ	JE ／JZ
A ≦B	JBE／JNA	JLE／JNG
A ＜B	JB ／JNAE	JL ／JNGE
A ≠B	JNE／JNZ	JNE／JNZ

＊CMP A，Bとした場合

例　題

バイトデータ11Hを100H番地に，88Hを101H番地に格納したうえで，これらの数値を，符号なし数値とみた場合に大きい方を102H番地に格納しなさい．また，これらの数値を符号付き数値とみた場合に大きい方を103H番地に格納しなさい．

考え方

フローチャートを書いてみましょう．(次ページに掲載)

ここで，符号なし条件分岐命令にはJAE命令，符号付き条件分岐命令にはJGE命令を用いればよいことがわかります．

ところで，アセンブルするうえで少し問題となることがあります．今までの条件分岐命令は，繰り返しのため前の番地に飛ぶものばかりでした．ところがここでは先の番地に飛ぶ命令がでてきます．先の命令の格納される番地はまだわかりません．さて，どうしたらよいでしょうか？

先のアドレスに分岐する方法

このような場合，BASICでは適当な行番号を付けるかまたは，ラベルを使用しました．たとえば，

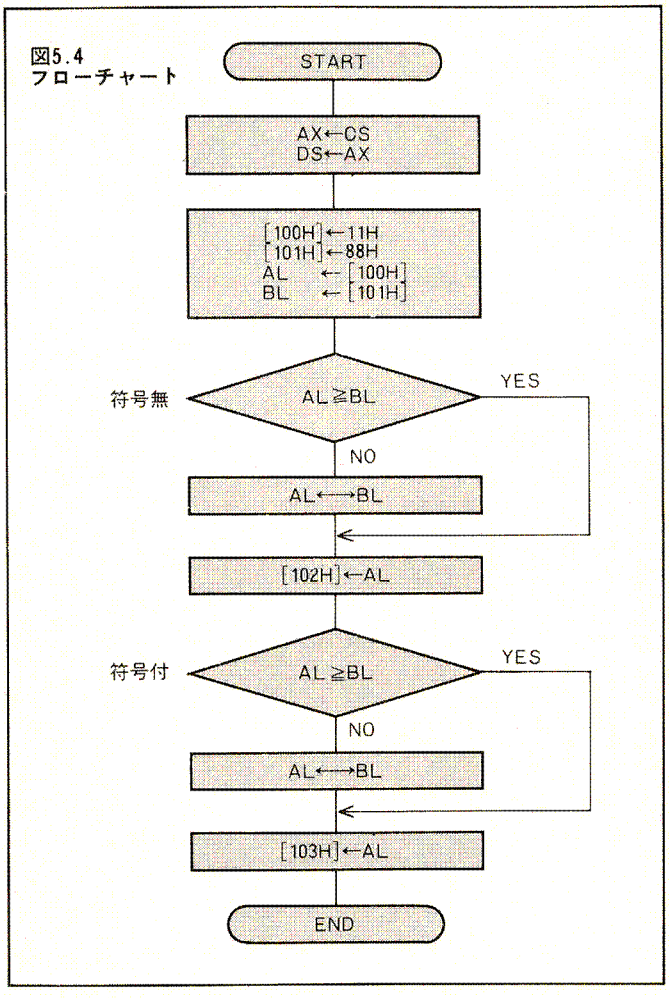
```
IF　A>=B　THEN　*LABEL1
```

などのように使いました．ところがアセンブラではそれができません．とはい

ってもASM86のようなアセンブラではラベルを使うことができるのですが，ここではPC-9801の内蔵モニタのアセンブラを使用することを前提としているので，それはできません．それではどうしたらよいのでしょうか．

以下にその入力の手順を示したので見て下さい．

図5.4
フローチャート




```
h]A0
0000 8CC8            MOV AX,CS↵
0002 8ED8            MOV DS,AX↵
0004 C606000111      MOV BYTE [100],11↵
0009 C606010188      MOV BYTE [101],88↵
000E A00001          MOV AL,[100]↵
0011 8A1E0101        MOV BL,[101]↵
0015 38D8            CMP AL,BL↵
0017 73E7            JAE 0↵
0019 86D8            XCHG AL,BL↵
001B A20201          MOV [102],AL↵
```

飛び先がわからないので，0番地にジャンプする命令として，かりに書いておきます．

```
001E 38D8              CMP AL,BL↵
0020 7DDE              JGE 0↵]①
0022 86D8              XCHG AL,BL↵
0024 A20301            MOV [103],AL↵
0027 F4                HLT↵
0028
h]A17↵
0017 7302              JAE 1B↵
0019
h]A20↵
0020 7D02              JGE 24↵
0022
```

飛び先が確定した段階で，個別にアセンブリし直します．フローチャートをよく見て指定する番地を間違えないように注意する．

意味はおわかりですか？この方法は面倒ですし，万能というわけではありませんが，内蔵アセンブラで先の番地へジャンプする命令を作成する場合には，この方法しかありません．

先にジャンプするには，一度書いてから書き直す

なお，条件分岐命令では機械語コードが必ず2バイトになるので問題はないのですが，機械語コードのバイト数が変化する可能性のある場合には注意が必要です．

解　答

それでは，入力結果を確認してみましょう．

```
h]L0,27
0000 8CC8              MOV     AX,CS
0002 8ED8              MOV     DS,AX
0004 C6060000111       MOV     BYTE [0100],11
0009 C606010188        MOV     BYTE [0101],88
000E A00001            MOV     AL,[0100]
0011 8A1E0101          MOV     BL,[0101]
0015 38D8              CMP     AL,BL
0017 7302              JAE     001B]── ALの方が大きければ，そのまま1BH番地に進みます．
0019 86D8              XCHG    BL,AL]─ 大きい方をALレジスタに入れます．
001B A20201            MOV     [0102],AL
001E 38D8              CMP     AL,BL
0020 7D02              JGE     0024
0022 86D8              XCHG    BL,AL
0024 A20301            MOV     [0103],AL
0027 F4                HLT
```

解　説

フローチャートと見較べながら，解説しましょう．

15H番地まではすぐわかりますね．17H番地ではALレジスタの方が大きいかまたは等しい場合，JAE(Jump if Above or Equal)命令ですぐにALレジスタの内容を格納する命令に進みます．もしBLレジスタの内容の方が大きい場合に

は，19H番地のXCHG命令でALとBLの内容を交換します．この命令の代わりに，

```
MOV     AL, BL
```

としてBLレジスの内容をALレジスタに転送してもいいのですが，そうすると，次に符号付きの大小比較をする場合，再びAL，BLレジスタに値を転送し直さなければならなくなり，面倒です．XCHG命令を使用すると内容は交換されるだけで，両方共残っていますから，次の処理が楽になります．

この部分がわかれば，次の符号付き数値として大小比較している部分もわかりますね．それでは，実行してみましょう．

```
hJG0,27↵
*1B00:0027↵
hJD100↵
0100  11 88 88 11 00 00 00 00 00 00 00 00 00 00 00 00          ||
hJ
```

符号なし数値としては88Hの方が大きく，符号付き数値としては11Hの方が大きいことがわかります．

4 無条件分岐命令

無条件分岐命令とは，その命令を実行すると必ず指定の番地にジャンプする，という命令です．BASICでいえばGOTO文に相当します．

例　題

符号なし数値0FFHを100H番地に，符号なし数値11Hを101H番地に格納し，その差を求め，100H番地の内容の方が大きいかまたは等しければ，その差の絶対値を102H番地に，101H番地の内容の方が大きければその差の絶対値を103H番地に格納しなさい．

考え方

問題文が少し面倒ですが，よく読んで下さい．この問題を考えるにあたって，次ページのようなフローチャートを考えました．

AL＜BLでない場合，すなわちAL≧BLのとき①の部分を実行するわけですが，それが終わった後，②の部分を飛ばしてENDに行かなければなりません．そこで，①の部分の終わりに無条件分岐命令を入れる必要があります．無条件分岐命令は，JMP（JuMP）と書かれます．このようなプログラムを作成する場合，フローチャートは**図3.5−2**のようになるのが普通ですが，ここでは流れがわかるようにたてに並べて書きました．

図5.5-1
フローチャート

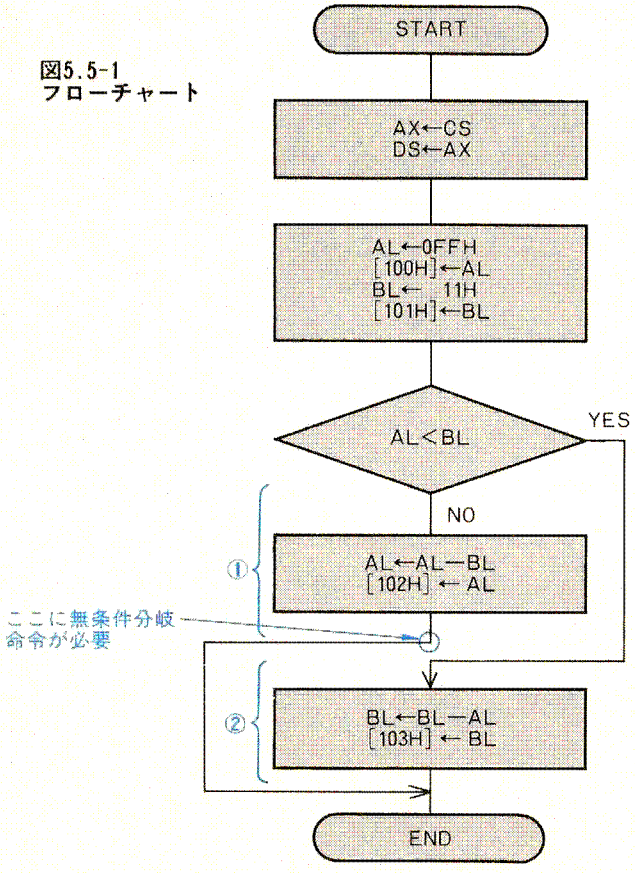
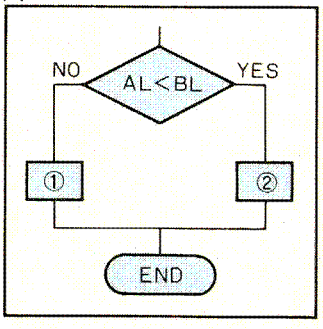


図5.5-2



今回も実際の入力法を示してみましょう．

```
hJA0 ↵
0000 8CC8          MOV AX,CS↵
0002 8ED8          MOV DS,AX↵
0004 B0FF          MOV AL,FF↵
0006 A20001        MOV [100],AL↵
0009 B311          MOV BL,11↵
000B 881E0101      MOV [101],BL↵
000F 38D8          CMP AL,BL↵
0011 72ED          JB 0↵
0013 28D8          SUB AL,BL↵
0015 A20201        MOV [102],AL↵
0018 E9E5FF        JMP 0↵
001B 28C3          SUB BL,AL↵
001D 881E0301      MOV [103],BL↵
0021 F4            HLT↵
0022               ↵
```

はじめは0としておきます．

```
hJA11
0011 7208              JB 1B↵
0013                   ↵
hJA18↵
0018 E90600            JMP 21↵
001B                   ↵
```

解　答

```
hJL0,21
0000 8CC8              MOV     AX,CS
0002 8ED8              MOV     DS,AX
0004 B0FF              MOV     AL,FF
0006 A20001            MOV     [0100],AL
0009 B311              MOV     BL,11
000B 881E0101          MOV     [0101],BL
000F 38D8              CMP     AL,BL
0011 7208              JB      001B
0013 28D8              SUB     AL,BL
0015 A20201            MOV     [0102],AL
0018 E90600            JMP     0021
001B 28C3              SUB     BL,AL
001D 881E0301          MOV     [0103],BL
0021 F4                HLT
```

解　説

はじめにデータを格納する方法を少し変えてみました．JMP命令が新しく加わった他は，とりわけ難しい点はないと思います．フローチャートさえきちんと書くことができれば，あとはそれを命令に直していくだけですから，問題はないでしょう．

実行結果は，103H番地に 0EEHが格納されますね．数値を変えて実験してみて下さい．

5 無条件分岐命令とアドレス指定法

ここまで，条件分岐命令，無条件分岐命令とも特に何も意識せずに，命令の中で分岐先のアドレスを指定してきました．今までのところはうまくいっていたのですが，実はちょっとした注意が必要です．

まず，**分岐先のアドレスを指定する方法で分類します．**大きく分けると3つになり，

直接アドレス指定法
レジスタポインタ間接アドレス指定法
メモリポインタ間接アドレス指定法

があります．

次に，**分岐先のアドレスの範囲の制限によって分類します．**これも3つに分けられます．

Short-label　（−128〜＋127内の範囲）
Near-label　（同一セグメント内）
Far-label　（別のセグメント内）

またこれは，初めのうちはあまり考慮しなくてもよいことですが，**分岐命令を機械語に変換する際の方法で分類すると，**

相対アドレス指定法
絶対アドレス指定法

の2つの方法があります．

たいへんだナァと思われましたか？3つの分類のうちの最初の分類法，すなわち**直接，レジスタ，メモリ**の3つの指定法さえ憶えておけば，あとの分類法は気にしなくともかまいません．

5−1　直接アドレス指定法

それでは，実際にどのように使用するのかを，**分岐先のアドレスを指定する方法**で分類し，説明していきましょう．

まず，直接アドレス指定法ですが，これはこれまでの例題のなかで使用してきた方法で，分岐先としてアドレスを数値で書く方法です．

```
JMP     1234H
```

のような指定法がそれです．この方法は分岐先のアドレスによりさらに3つに分けられ，現在実行中の命令を起点として，−128〜＋127バイトの範囲に分岐する場合には，

アドレスを数値で指定する方法

```
1200  EB32          JMPS    1234H
```

という命令が使用できます．この命令はJMP命令に比べて，翻訳されてできる機械語が1バイト短かくて済む，という利点を持っています．ただし，分岐先の範囲に注意しなければなりません．

現在実行中の命令を起点として，同一セグメント内に分岐する場合には，先に述べたように，

```
1000  E93102        JMP     1234H
```

というような書き方をします．なお，ここでオペランドとして指定する番地は，分岐先のオフセットアドレスです．同一セグメント内であれば範囲を考慮する必要はありません．生成する機械語は相対アドレスとなりますから，まったく同じ命令を別のアドレスに置くと違った機械語を生成します．しかしこのことは今は特に考慮しなくてかまいません．

相対アドレス

もうひとつの直接アドレス指定法を用いた命令に，セグメント間分岐命令があります．この方法は，分岐先のアドレスとして新しいコードセグメントの値と，そのコードセグメント内でのオフセットアドレスを指定する方法です．

しかしPC-9801の内蔵モニタのアセンブラでは，セグメント間ジャンプの命令は使用することができませんので，この命令はアセンブルできません．もし仮りに書くとしたら，次のような形式になるかも知れませんが，これはあくまでも想像です．

セグメント間分岐はアセンブルできない

```
JMPF    1234H, 5678H   ??
```

ASM86などのアセンブラでは，一応セグメント間分岐命令もアセンブルできますが，この命令はめったに使用することはありませんし，あまり使用しない方がよい命令ですから，皆さんもこの命令については知らなくてもかまいません．セグメント間でのジャンプなどには通常，INT命令を用います．（6章参照）

5-2　間接アドレス指定法

次に，**レジスタポインタ間接アドレス指定法**を用いたJMP命令の使用法について説明します．この方法は，レジスタに，分岐先のアドレスを数値として代入しておき，そのレジスタの内容に従って分岐する，という方法です．具体的には，

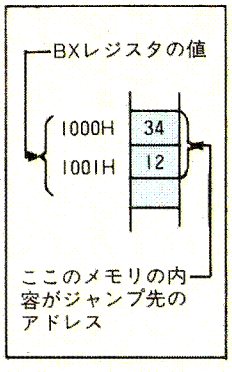
レジスタを用いたアドレス指定

```
JMP     AX
```

のように使用します．したがって，レジスタにはあらかじめ値を代入しておかなければなりません．上述の例では，AXレジスタの内容が1234Hであったとすると，1234H番地に分岐します．この指定法で使用できるレジスタは16ビットレジスタ，すなわちAX，BX，CX，DX，SP，BP，SI，DIの各レジスタです．この命令では，同一セグメント内へ自由に分岐できます．

メモリを用いたアドレス指定

最後に，**メモリポインタ間接アドレス指定法**を用いたJMP命令の使用法について述べましょう．メモリポインタ間接アドレス指定法では，あらかじめメモリに分岐先のアドレスを書き込んでおきます．そして，そのアドレスの書き込まれている番地を指定することにより，分岐先を決定します．具体的には，




```
JMP     [BX]
```

のように使用します．この場合，たとえば1000H，1001H番地に数値1234Hを，逆ワード形式で格納しておき，BXレジスタに1000Hを代入してからこの命令を実行すると，1000H，1001H番地の内容である1234Hを読み出して，1234H番地に分岐することになります．

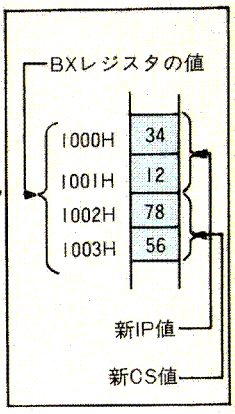
この場合，間接指定の方法は，「MOV命令」のところで解説した間接指定の方法とまったく同じですから，「MOV命令の整理」の部分を参照して下さい．

そこで，たとえば，

```
JMP     1234〔BX+SI〕
JMP     〔1234〕
```

といった使用法ができます．2つめの例は，間接指定の特殊な場合である．ダイレクト指定でしたね．

メモリポインタ間接アドレス指定法では，もうひとつ，セグメント間ジャンプ命令も使用できます．この方法は，あらかじめメモリ上に分岐すべきアドレスのオフセット値と，セグメントベース値を書き込んでおき，JMP命令の中で，そのメモリの番地を指定する方法です．書き方としては，




```
JMPF    [BX]    ??
```

のようになるはずですが，これも内蔵モニタのアセンブラでは翻訳することはできません．

一応使用例を示しておきましょう．まずメモリの1000H，1001H番地に1234Hを，1002H，1003H番地に5678Hを格納したうえで，BXレジスタに1000Hを代入し，上記の命令を実行すれば，新しいコードセグメントに5678Hを設定したうえで，そのコードセグメント上の1234H番地に分岐します．

なお，上記のような表記法は，使用するアセンブラによって違う形式となっていますし，前述したように，セグメント間の分岐はあまり使用しない方が良いので，この例も参考程度に考えて下さい．

6 条件分岐命令のアドレス指定法

条件分岐命令（JAE，JBなど）での分岐先のアドレス指定法は，ただ1通りしかありません．それは無条件分岐命令の最初に述べた方法で，**直接アドレス指定法**で，しかもその命令を起点として−128〜＋127バイトの範囲への相対分岐のみです．したがって，使用法は

条件分岐命令はショートジャンプのみ

```
0000    7318    JAE    001A
```

などのような方法だけです．

それでは，同一セグメント内のかなり離れたアドレスに分岐するには，どうしたらよいでしょうか？この場合は，一度条件分岐命令で適当なアドレスに分岐し，その分岐先にさらに無条件分岐命令を書いておいて，遠くへ分岐する，という方法をとります．若干面倒な方法ですが，8086CPUでは，この方法をとらざるを得ません．

7 相対分岐型と絶対分岐型

7－1　相対分岐法

今まで解説してきたなかで，その命令を起点として－128～＋127バイトの範囲，といった制限のあるものがありました．それは，分岐先のアドレスを指定する方法として相対分岐法を用いている命令です．相対分岐法とは，現在実行中の命令のあるアドレスを中心として，分岐先がどれだけ離れているか，という形で機械語の命令を作る方法です．

次の例を入力してみて下さい．

```
h]A1000 ↵
1000    E90D00      JMP     1010
1003
```

ここで，JMP命令のところで翻訳された機械語に注意して下さい．[E9] [0D] [00] となっていますね．この命令ではじめの [E9] は，ジャンプ命令を表わす機械語のコードです．次の [0D] [00] が飛び先を指定する部分ですが，アセンブラでは1010と入力したにもかかわらず，0D00となっていますね，逆ワード形式ですから，ここの数値は実際は000Dとなります．この理由について考えてみましょう．

相対分岐の機械語コード

命令を順番に実行していく場合，その命令の存在するアドレスはIP（インストラクションポインタ）レジスタに保持されています．CPUはIPの示すアドレスから，1バイトの機械語を読み込みます．この例でいえば，1000H番地の [E9] を読み込みます．するとこの機械語コードは無条件分岐命令であることがわかります．そしてその後の2バイトが，分岐先の情報であることがわかりますから，これもいっしょに読み込みます．このとき命令の長さが3バイトであることがわかりますから，CPUはこの命令を読み込んだ直後，もとのIPの値1000Hに3Hを加えて，IPが次の命令の書かれている番地である1003Hを指すようにします．

さてそのうえで，CPUは分岐命令を実行するために分岐先のアドレスの計算を始めます．このときCPUは，現在のIPの値（＝1003H）に，命令のなかに書かれていた数値（＝000DH）を加えます．そしてこの加えた値（1003H＋000DH＝1010H）をIPに格納し直します．するとIPの値は1010Hになりますね．次にCPUが新しい命令を実行する場合は，IPの示すアドレスから命令を読み込むわけですから，1010H番地の命令を実行することになります．つまり1010H番地へジャンプしたことになるわけです．

相対アドレス計算法

以上のことからわかることは，相対分岐のアドレスは，その分岐命令の次の命令のアドレスからどれだけ離れているかで計算される，ということです．

処理の流れは次ページのようになります．

図5．6　処理の流れ

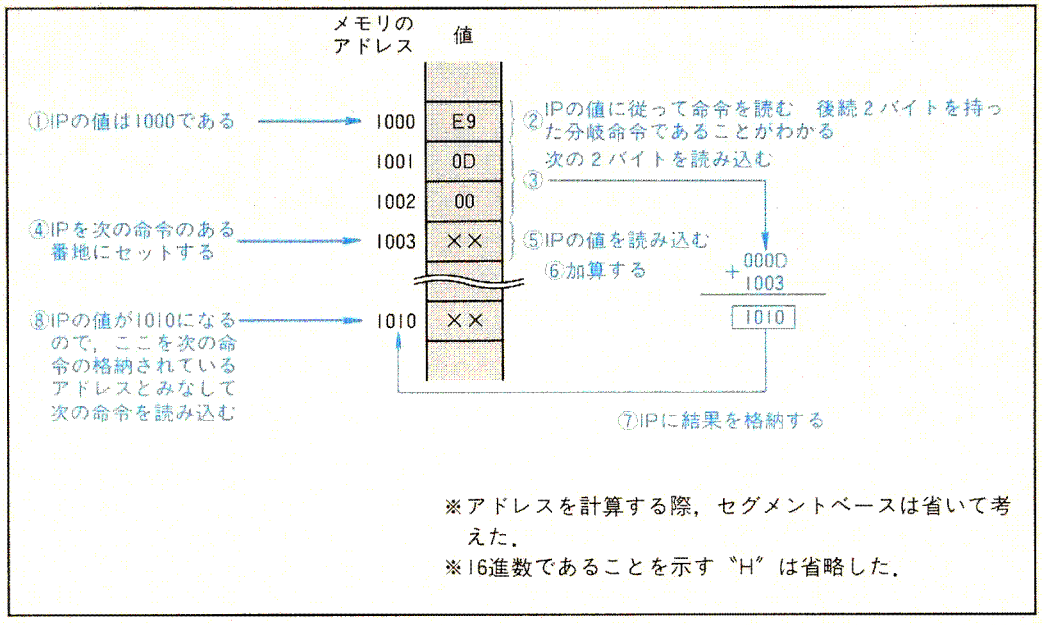
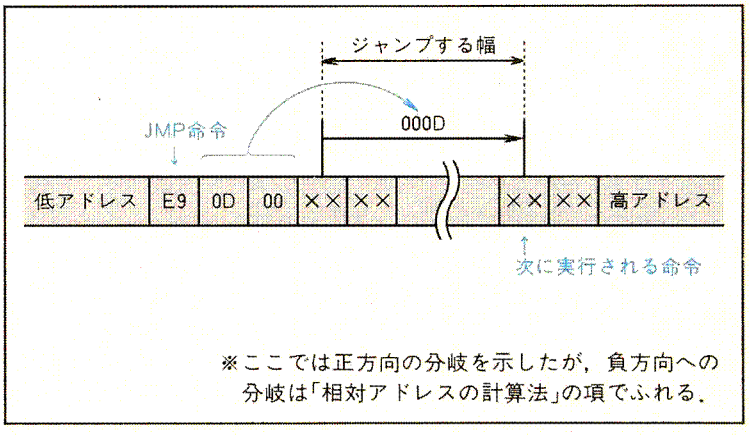


図5．7
アドレスの計算



上記のようなアドレスの計算は特に考慮しなくても，アセンブラが翻訳の時点で自動的に計算してくれますが，アドレスの指定法として一応憶えておいて下さい．なお，さらに詳しいことは，命令の整理のあとの「相対アドレスの計算法」で説明します．

7－2　絶対分岐法

絶対分岐型

それでは，絶対分岐型でのアドレスの計算法について説明しましょう．絶対分岐型では，その命令のあるアドレスにかかわりなく，レジスタの内容あるいはメモリの内容に従って分岐します．たとえば，

```
MOV     AX, 1000
JMP     AX
```

という命令を実行すると，現在の命令のアドレスがどこであろうと，命令実行後，1000H番地に分岐します．また，

相対分岐はアドレスを移動しても一緒に移動する

```
MOV     WORD [1000], 1234
JMP     [1000]
```

では，この命令がどこにあっても1000H番地の内容（＝1234H）を取り出してきて，1234H番地に分岐します．このように命令の格納されているアドレスにかかわらず，分岐先のアドレスが決まるので，このようなタイプの分岐命令を絶対分岐型命令といいます．一方，相対分岐型では前ページの図を見ると明らかなように命令が0000H番地にある場合は0010H番地に分岐し，1000H番地にある場合には1010H番地に分岐する，というように命令の格納される場所に従って，分岐先がいっしょに移動してしまう点が異なっていますね．

なお，命令が相対分岐型か絶対分岐型かは，命令の種類によって決まっています．したがって相対分岐型と絶対分岐型は命令に拘束され，自由に使い分けることはできません．同じ範囲の分岐命令を相対型と絶対型の両方で使い分けることのできる68系などのCPUを知っている人にとっては，このあたりが不便に思われる点かもしれません．

相対か，絶対かは命令の種類で決まる

それでは，しめくくりとして分岐命令のアドレス指定法を整理しておきます．

分岐命令の整理　①分岐先の範囲と指定法

		アドレス指定法	分岐先アドレス計算法	分岐範囲（オフセット値）	使用法
無条件分岐命令	セグメント内	直接（Short-label）	相対	−128〜＋127*	JMPS 1234
		直接（Near-label）	相対	同一セグメント内*	JMP 1234
		レジスタポインタ間接	絶対	同一セグメント内	JMP AX
		メモリポインタ間接	絶対	同一セグメント内	JMP [BX]
	セグメント間	直接（Far-label）	絶対	全メモリ空間	———
		メモリポインタ間接	絶対	全メモリ空間	———
条件分岐命令	セグメント内	直接（Short-label）	相対	−128〜＋127*	JAE 1234

＊本節末「相対アドレスの計算法」参照．

JMPS　即値 ……即値で示すアドレスに分岐する．分岐先は−128〜＋127バイトの範囲内に限られる*

JMP　即値 ……即値で示すアドレスに分岐する．同一セグメント内への分岐が可能*

JMP　reg16 ……レジスタの内容の示すアドレスに分岐する．

JMP　mem16……メモリの内容の示すアドレスに分岐する．

Jc　　即値　……cで示す条件が満たされた場合に，即値で示すアドレスに分岐する．分岐先は－128～＋127バイトの範囲内に限られる．条件が満たされなかった場合には，後続の命令を実行する．(Jcとして使用できる命令は，以下の2つの表に掲げる．)

＊本節末「相対アドレスの計算法」参照

＊セグメント間分岐命令は、内蔵モニタのアセンブラでは使用できないので省略した．どうしても使用したい人は，CP／M-86，上のASM86などを使用するか，または，巻末の表を用いてハンドアセンブルすること．

分岐命令の整理　②比較により分岐する命令

A，Bの大小関係	符号無し数値の場合	条件を満たすフラグの状態		符号付き数値の場合	条件を満たすフラグの状態	
		CF	ZF		SF ∀ OF	ZF
A＞B	JA ／JNBE	0	0	JG ／JNLE	0	0
A≧B	JAE／JNB	0	0	JGE／JNL	0	×
A＝B	JE ／JZ	×	1	JE ／JZ	×	1
A≦B	JBE／JNA	1 or	1	JLE／JNG	1	1
A＜B	JB ／JNAE	1	×	JL ／JNGE	1	×
A≠B	JNE／JNZ	×	0	JNE／JNZ	×	0

※大小関係は，CMP　A，Bで比較した場合である(A－Bの結果)

※SF ∀ OFとは，SF（サインフラグ）とOF（オーバーフローフラグ）の排他的論理和を表わす．すなわち，どちらか一方のみが1のときに1となり，両方とも0かまたは，両方とも1のとき0となる．

※×印の欄は0でも1でもかまわないことを意味する．

※JA／JNBEのように，／で区切られた2つの表現は，まったく同じ関係を別の言葉で表現したものである．アセンブラではどちらを使用してもよい．ただし逆アセンブルした場合は，すべて左側の表現に統一される．

分岐命令の整理　③直接フラグまたはレジスタによって分岐する命令

フラグ	1のとき分岐	0のとき分岐	フラグが1のときの意味
SF	JS	JNS	補数表現で負の数である．
ZF	JZ／JE	JNZ／JNE	結果が0である（上表のA＝Bと同じ）．
PF	JP／JPE	JNP／JPO	1となったビットの数が偶数個である．
CF	JB／JNAE	JNB／JAE	キャリーフラグが1である（演算の結果キャリーまたはボローが生じた場合など）．
OF	JO	JNO	補数表現を用いた演算で結果が範囲を超えた．
レジスタ			命 令 の 意 味
CX	――――	JCXZ	CXレジスタが0のときだけ分岐する．

※これらの命令は，条件分岐命令と同じく，分岐先が－128～＋127バイトの範囲に限られる．次の「相対アドレスの計算法」参照せよ．

8 INC命令・DEC命令

INC命令（INCrement）は，指定されたレジスタまたはメモリの内容を1増加させます．DEC命令（DECrement）は，指定されたレジスタまたはメモリの内容を1減少させます．

フラグは結果に従って変化しますが，キャリーフラグだけは変化しません．（INC　ALという命令で，もとのALの内容が0FFHだったとすると，新しいALの内容は0Hになりますが，キャリーフラグは変化しません．）

なお，メモリを指定する場合は，必ずBYTEまたはWORDの指定が必要です．

INC命令・DEC命令の整理

```
INC     reg      ……レジスタの内容を1増加させる．
INC     mem *……メモリの内容を1増加させる．
DEC     reg      ……レジスタの内容を1減少させる．
DEC     mem *……メモリの内容を1減少させる．
```

※この命令群では結果に従って，CFを除くフラグが変化します．

9 比較命令（CMP）

CMP　A，Bは
A－B⇨フラグ

比較命令（CMP：CoMPare）は，基本的には減算命令（SUB）と同じ働きをします．すなわち，第1オペランドから第2オペランドの内容を減じます．しかし，減算命令と違って，結果を第1オペランドに格納せず，フラグレジスタの内容だけを変化させます．大小関係のみを比較し，レジスタまたはメモリの内容を変化させたくない場合に使用します．

比較命令の整理

```
CMP     reg，即値      ……レジスタの内容と即値を比較する．
CMP     mem，即値  *……メモリの内容と即値を比較する．
CMP     reg，reg       ……レジスタの内容どうしを比較する．
CMP     mem，reg       ……メモリの内容とレジスタの内容を比較する．
CMP     reg，mem       ……レジスタの内容とメモリの内容を比較する．
```

※この命令群では結果に従ってフラグが変化する．

それでは，長々と続いた文法解説に少々退屈されている方もいると思いますので，本章の結びとして練習問題を2題ほどやってみましょう．

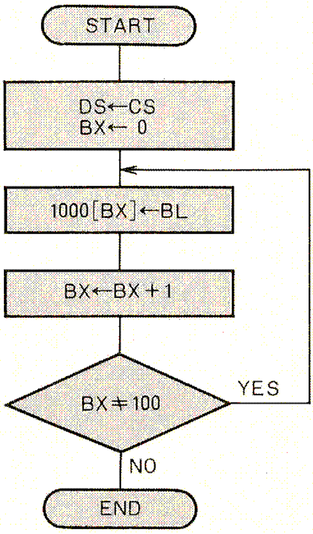
練習問題

5－1

1000H番地に00Hを，1001H番地に01Hを，1002H番地に02Hを，以下同様にして10FFH番地までを順に数値で埋めなさい．

考え方

図5.6 フローチャート



フローチャートは，左図のようになります．

解答

```
0000 8CC8        MOV    AX,CS
0002 8ED8        MOV    DS,AX
0004 BB0000      MOV    BX,0000
0007 889F0010    MOV    1000[BX],BL
000B 43          INC    BX
000C 81FB0001    CMP    BX,0100
0010 75F5        JNE    0007
0012 F4          HLT
```

［JNAE，JBを用いてもよい．

解説

これはやさしすぎましたね．解答のように，BXレジスタを，アドレスの指定とデータの両方に用いる方法が最も効果的なプログラムですが，別々のレジスタを用いて指定してもかまいません．なお，今回はBXの値を増加させる方向で

データを転送しましたが，減少させて0になったかどうかを判別する方法を用いる場合には注意が必要です．悪例として次のプログラムを見て下さい．

```
0000 8CC8          MOV    AX,CS
0002 8ED8          MOV    DS,AX
0004 BBFF00        MOV    BX,00FF
0007 889F0010      MOV    1000[BX],BL
000B 4B            DEC    BX
000C 81FB0001      CMP    BX,0000
0010 75F5          JNE    0007
0012 F4            HLT
```

なぜ良くないのかわかりますか？この方法では，BXが0になるとすぐに終了してしまい，1000H番地に値00Hを転送せずに終了するからです．この場合には000CH番地のJNEをJGEに変更すれば，正しく動作しますが，この使用法にも注意が必要です(注)そこで，次のように変更してみます．

```
0000 8CC8          MOV    AX,CS
0002 8ED8          MOV    DS,AX
0004 BBFF00        MOV    BX,00FF
0007 889F0010      MOV    1000[BX],BL
000B 4B            DEC    BX
000C 83FB00        CMP    BX,0000
000F 75F6          JNE    0007
0011 F4            HLT
```

これが減少法による正しいプログラムです．このプログラムの000CH番地のCMP BX，0000という命令は省略することができます．それは0007番地でDEC　BXを実行した結果のフラグが，次のMOV命令では変化せずに残っているからなのです．MOV命令ではフラグが変化しない，ということは憶えておいて下さい．

JGEを使うときは負の数に注意

注）仮に0FFFFH個のデータを転送するとする．BXに0FFFFHを代入して1回実行し，JGEまできた時この数は負の数と解釈されるので，実行が終了してしまい，データの転送が1個しかできない．JAE命令では永久に終了しないので使えない．

実行結果は次のようになります．

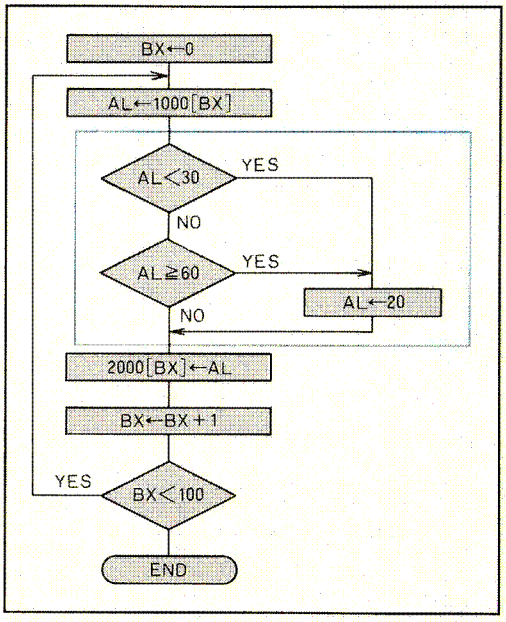
```
h]G0,12
*1B00:0012
h]D1000,10FF
1000 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
1010 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
1020 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F    !"#$%&'()*+,-./
          32 33 34 35 36 37 38 39                        56789:;<=>?
1090 90                               9C 9D 9E 9F
10A0 A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF    .「」、・ヲァィゥェォャュョッ
10B0 B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF    ーアイウエオカキクケコサシスセソ
10C0 C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF    タチツテトナニヌネノハヒフヘホマ
10D0 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF    ミムメモヤユヨラリルレロワン゛゜
10E0 E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF    =田▲▼♠♥♦♣●○/
10F0 F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF    ×円年月日時分秒
```

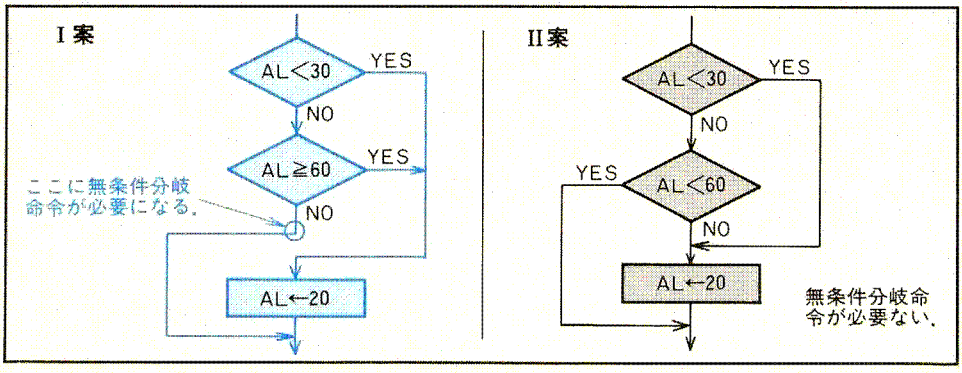
5－2

前問の実行の結果1000H番地～10FFH番地に00H～0FFHが格納されている．これを2000H番地以後に転送しなさい．ただしその際に，転送する数値が30H以上，60H未満ならそのまま格納し，それ以外であれば20Hを格納しなさい．(前問のプログラムに続ける形で書きなさい．)

考え方

さて，問題はワクで囲った部分ですが，この部分をプログラムするにはどうしたらいいでしょうか？プログラムはあくまでも一本の連続した流れですから，このように2種類の処理に分かれた場合は，次のようにしなければなりません．





I案は，はじめのフローチャートをそのまま位置をずらして書いたものです．しかしこの方法では，AL≧60の判断の結果NOとなった場合，さらに無条件分岐命令でAL←20の後へ分岐しなければなりません．

II案は，まったく同様のことを判断の条件を変えることにより，分岐の回数を減らして実現したものです．こちらの方がすぐれていることは，もうおわかりですね．それでは，はじめのフローチャートの囲み内の部分を，このII案と入れ換えてプログラムを作成してみて下さい．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 BB0000        MOV     BX,0000
0007 889F0010      MOV     1000[BX],BL
000B 43            INC     BX
000C 81FB0001      CMP     BX,0100
0010 75F5          JNE     0007 ←――――――――――[(JBでもよい).
0012 BB0000        MOV     BX,0000
0015 8A870010      MOV     AL,1000[BX]
0019 3C30          CMP     AL,30
001B 7204          JB      0021
001D 3C60          CMP     AL,60
001F 7202          JB      0023
0021 B020          MOV     AL,20
0023 88870020      MOV     2000[BX],AL
0027 43            INC     BX
0028 81FB0001      CMP     BX,0100
002C 72E7          JB      0015 ←――――――――――[(JNEでもよい).
002E F4            HLT
```

解　説

実行結果は次のようになります.

```
hJG0,2E
*1B00:002E
hJD2000,20FF
2000 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
2010 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
2020 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
2030 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F    0123456789:;<=>?
2040 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F    @ABCDEFGHIJKLMNO
2050 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F    PQRSTUVWXYZ[¥]^_
2060 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
2070 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
2080 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
2090 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20A0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20B0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20C0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20D0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20E0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20F0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
```

フローチャートが書ければあとは簡単ですね．しかし，はじめは完璧なフローチャートを作成することは難しいと思います．特に，レジスタの上手な使い方などはなかなかできないものです．

フローチャートを作るとき，はじめはレジスタのかわりにA，B，C……またはa，b，c……などを用いて作成し，あとで最も適当なレジスタをあてはめていけばよいと思います．どうしてもレジスタが足りなくなったときは，メモリを使用しなければなりません．また，同じレジスタを16ビットと8ビットの両方に使い分けるのは難しいことですから，この問題でもはじめはBLレジスタではなくCL，DLなどのレジスタを使用してプログラムを作成してみたうえで，検討するとよいでしょう．

10 相対アドレスの計算法

相対分岐型のアドレス指定法では，アドレスの計算を次のような方法で行ないます．たとえば，1000H番地に次の命令が書かれていたとします．

```
1000    E9[FDDF]      JMP    F000
[1003]     └──変位
 └──新IP値
```

このとき，1000H番地の命令を解読したCPUは，新しいIP（インストラクション・ポインタ）の値である1003Hに，命令の機械語中に書かれている変位の値である 0DFFDHを加えます．

相対アドレスは命令の次のアドレスから計算される

```
  IP──→     1003H
変位──→    0DFFDH
─────────────────
           0F000H──→分岐先  0F000H
```

そして上記のように，結果として0F000Hを分岐先のアドレスとして得ます．

さて，それでは分岐先として0100H番地を指定した場合にはどうなるでしょうか？

```
1000  E9FDF0      JMP      0100
1003
```

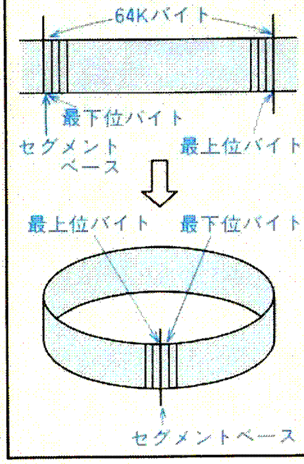
この場合にもCPUは，実は同じように計算を行なっています．

```
  IP──→     1003H
変位──→    0F0FDH
─────────────────
          10100H──→分岐先  0100H
          └桁上がり（無視する）
```

この場合，加算の結果として得られた桁上がりを無視し，下位の4桁の値を分岐先のアドレスとします．つまりアドレスの計算においては，見かけ上0FFFFH番地の上に0H番地があるようになるわけです．このことは図5.7のように，アドレスが輪のようになっている，と考えるとわかりやすいでしょう．CPUは，このような計算の結果得られたIPの値（オフセット値）に，セグメントベースの値を加えて，絶対番地（物理アドレス）を生成しています．そのため，この方法によってセグメントのワクからはみ出すことなく，同一セグメント内のアドレスをすべて指定できることになります．

アドレスの計算は輪になっている

図5.7　アドレス



この方法は，命令のあるアドレスより前にあるアドレスを指定できるため，あたかも符号付き数値で分岐先を指定しているように見えます．しかも符号付き数値では指定できる範囲に制約があるのに比べ，この方法では，同一セグメント内をすべて指定できます．したがって，ここで使用される数値は符号付き数値とは若干ニュアンスが異なります．一部の解説書は，相対アドレスの範囲が－32768d～＋32767dであるとしていますが，これは誤りです．

それではショートラベルの場合，すなわちJMPS命令や条件分岐命令において，分岐先の範囲が－128d～＋127dとなっているのは符号付き数値ではないのかというと，実はこれも符号付き数値とは若干ニュアンスが異なり，同じ方法で計算されているのです．つまり16進2桁の変位の値を4桁に符号拡張したうえで，上記と同じ方法でアドレスの計算をしているのです．したがって，次のような命令を実行することが可能です．

```
0000  77 [EE]        JA     FFF0
[0002]      ↑―変位
 └新IP値
            IP――――――――――→  0 0 0 2H
            符号拡張された変位――→0 F F E EH
                                ―――――――――
                                0 F F F 0 H ――→分岐先  0FFF0H
```

この命令を実行すると，0番地から0FFF0H番地へ分岐することができます．

なお，「アドレッシング・モード」で述べたディスプレースメントも，同様の方法で計算されます．したがって，たとえばBX＝2000Hとしたとき，

F000[BX]

と書いた場合には，

```
            BX ――――――――――→ 2000H
            ディスプレースメント――→ F000H
                                 ――――――――
                                [1]1000H――――→1000H
                             無視―┘
```

となり，1000H番地を指定することになります．8ビットのディスプレースメントについても，前記と同様符号拡張して加算されるため，あたかも符号付き数値であるように使用されています．

なお，負のディスプレースメントをアセンブルする場合は次のようにします．

```
h] AO ⏎
0000  8B8700F0      MOV    AX, 0−1000[BX] ⏎
```

この場合BXレジスタの値より，1000H番地下位にあるアドレスの内容がレジスタに送られます．逆アセンブルしてみると，

```
h] L0, 3 ↵
0000  8B8700F0      MOV     AX, F000 [BX]
```

負のディスプレースメントの計算法
0－と
100－とは違う

となっています．なお，これは16ビットディスプレースメントを持つ場合です．8ビットディスプレースメントにおいては，たとえば10Hだけ下位にずらした場合には，

```
h] A0 ↵
0000  8B47F0        MOV     AX, 100－10 [BX] ↵
```

ここを100にすると，8ビットのディスプレースメントが生成される．0のとき16ビットディスプレースメントになる．

のようにします．続けて次のように入力してみて下さい．

```
0003  8B87F000      MOV     AX, 0－10 [BX] ↵
```

どうです？生成された機械語が違うでしょう．この例は，わざわざ16ビットのディスプレースメントとして計算してしまったわけです．上の2つの機械語は，見かけは違いますが，まったく同じ動作をします．

例　題

次の分岐命令の分岐先の番地を，機械語コードから計算せよ，(どの場合も，はじめの1バイトが命令の種類を，それに続くバイトが変位を表わす)

0000	E9FD7F	JMP	□
F000	E900F0	JMP	□
0010	EBE0	JMPS	□
8050	E9AD7F	JMP	□
FFF0	735E	JAE	□
FFF0	73F0	JAE	□

解　答

計算式と共に横に並べて示します．

0003	0F003	0012	8053	0FFF2	0FFF2
+7FFD	+0F000	+0FFE0	+7FAD	+　5E	+0FFF0
8000	1E003	0FFF2	10000	10050	1FFE2
8000H	0E003H	0FFF2H	0H	50H	0FFE2H

2 BASICで機械語を

ここまで読み進めてきた皆さんなら，簡単なプログラムならマシン語で組めるようになったことでしょう．しかし，いくらマシン語でプログラムを作っても，データの入力や結果の出力ができなければ，効果は半減してしまいます．

そこでこの節では，BASICで入出力を行ない，主要な計算のみをマシン語で行なう方法について学んでいきましょう．

1 メモリへの書き込み・読み出し（PEEK・POKE文）

PEEK・POKE文は，

A＝PEEK(ADR)

と表わし，変数ADRで指定する番地の内容（1バイト）を読み出して，変数Aに代入します．読み出された数値は1バイトなので，実行の結果Aは0～255までの間の整数になります．

POKE文は，

POKE ADR, A

と表わし，変数ADRで指定する番地に，変数Aの値を格納します．Aは，0～255までの間の整数でなければなりません．

PEEK，POKEともに，ADRは0～65535（0H～0FFFFH）の間の整数でなければなりません．ここで指定されるアドレスは，DEF SEG文によって指定されたセグメントベースからのオフセットアドレスです．したがってPEEK，POKE文を使用する前には，必ずDEF SEG文でセグメントベースを指定しておかなければなりません．

コマンドと関数の違い

マニュアルでは，PEEKは関数の項に，POKEはコマンドの項に記載してあるので，不思議に思われた方が多いと思いますので，一言説明しておきましょ

う．

PEEKは，ある値を与えると結果として数値を与えるので関数と呼ばれ，POKEは，実行の結果としての数値が得られないのでコマンドと呼ばれます．このことはDSKIで，DSKO$についても言えますし，あとに述べるUSR，CALLも同様です．

では例題をやってみましょう．

例　題

キーボードから入力した数値（0～65535の範囲の整数）をセグメントベース1B00H，オフセットアドレス100H，101HにPOKE文を使って格納し，すぐにPEEK文で読み出して表示しなさい．入力された数値が範囲を越える場合は，エラーメッセージを出力して終了しなさい．

解　答

```
100 DEF SEG=&H1B00
110 INPUT"INPUT NO.";A
120 IF A<0 OR A>65535! THEN PRINT"ERROR":END
130 'POKE &H100,A MOD 256:POKE &H101,A¥256
140 A1=INT(A/256):A2=INT(A-A1*256):POKE &H100,A2:POKE &H101,A1
150 B=PEEK(&H100)+PEEK(&H101)*256
160 PRINT B
170 END
```

解　説

POKE文で書き込める数値は1バイトの数値です．0～65535の範囲の数値は2バイト分ありますから，この場合には上位バイトと下位バイトに分けて書き込む必要があります．その方法としては，130行のようにMODと¥を使う方法と，140行のように通常の乗除算を用いる方法とがあります．BASICの文法書によるとどちらでもいいような気がしますが，この2つの方法はかなり違うので次に説明しておきましょう．

MODと¥とINTの違い

①　MODと¥を用いる方法

まず，MODと¥を用いた方法ですが，MODは〝剰余〟を求める関数であり，MODの前に書かれた数値をMODの後に書かれた数値で割った余りを与えます．**この場合に使用できる数値は，－32768～32767の間の数値で，**最初に（絶対値で）四捨五入されてから計算されます．¥は商を求める関数で，同様に－32768～32767の間の数値で，（絶対値で）四捨五入されてから計算されます．

MOD，¥の範囲は－32768～＋32767

②INT関数と乗除法を用いる方法

INT関数と通常の乗除法を用いた場合，特に注意のない限り，Aは実数型数値として扱われ，A1の値として，Aを256で割った商を超えない最大の整数が与えられます．厳密性を抜きにして言えば，扱える数値の範囲は相当広くなります．詳しくは「BASIC　REFERENCE　MANUAL」に載っていますので，よく読んで下さい．特にINTとFIXとCINTの違いについてよく参照して下さい．また，前述のMODと¥の使用範囲の制限は，マニュアルに記載されていないので注意して下さい．

INTは切り捨てて小さな数に
FIXは切り捨てて0に

さて，そこで例題の場合入力される数値は0～65535の範囲なので，通常の乗除算を用いた140行の計算の方を使用するのが良いことがわかります．

MOD

130行のようにMODや¥を使用した場合，負の数についても商と余りを求めることができるのですが，残念ながらPEEK，POKE文に与えることのできる数値は正の数に限られています．したがって，130行のような使用法の場合には使える数値の範囲は，MOD文，POKE文の両方の制限によって0～32767までに限られることになります．扱う数値がこの範囲に入っている場合にはもちろん問題はないのですが，例題の場合には使用できません．また，負の数値を(補数表現を用いて）メモリに格納しようとする場合には，もう少し複雑な計算が必要となることがおわかりいただけると思います．

実行結果についてはあえて示す必要もないと思いますので，自分で確かめて下さい，なお，メモリへの格納方法は逆ワード形式としましたが，このことにこだわる必要はありません．

DEF　SEG文の書き方注意！

もうひとつ蛇足ながら付け加えておくと，DEF　SEG文は必ずDEFとSEGの間を1文字（以上）あけて下さい．DEFSEG＝&H1B00というふうに続けて書くと，DEFSEGという変数に数値&H1B00が代入されてしまいます．

2 機械語ルーチンの呼び出し(USR関数)

BASICから機械語ルーチンを呼び出す方法としては，USR関数を使用する方法とCALLコマンドを使用する方法があります．本節では，とりあえず，BASICから機械語ルーチンを呼び出して計算をさせる方法をマスターしていただけるように，USR関数を使用した最も簡単な方法について解説します．

USR関数を使用するためには，次のような手続きが必要です．

```
DEF　SEG＝&H1B00←――セグメント・ベースの指定
DEF　USR0＝0←――――USR関数開始のオフセットアドレスの指定
```

DUMMY＝USR 0 (0)←――――USR関数の呼び出し

1行めで，機械語の入っているセグメントのベースアドレスを指定します．2行めでは，USR関数の開始アドレスを，セグメントベースからのオフセットアドレスで指定します．この場合には，0番地から始まるように設定しています．なお，USR関数は一度に10個まで定義することができ，USR0～USR9までを使用することができます．なお，USR 0の代わりに，単にUSRと書くこともできます．この文もDEF　USR 0のように，1文字分空白をあけなければなりません（逆にDEFINTなどは空白をあけてはいけません ）．

USR関数は10個まで定義できる

3行めでは，USR関数を呼び出しています．関数ですから，引数としてある数値（または文字）を与え，結果として数値（または文字）が与えられます．そのため上記のような書き方をします．

USR関数には，必要がなくても数値を代入

本来は与えた数値などは，機械語ルーチンに特別な方法で引き渡され，その計算結果を引き出すことができるのですが，少し複雑になるので本節では機械語ルーチンとの数値のやりとりはすべてPEEK，POKE文を使用することにします．そのためUSR関数に与える数値（この場合は0）と，計算結果（この場合はDUMMYという変数）は意味を持ちませんが，形式上必要なのでこのようにしておきます．

上述の3つの行は，間に別のプログラムが入ってもかまいませんが，必ずこの順序で置かれなくてはなりません．また，当然ですがUSR関数を使用する前に，機械語ルーチンがロードされていなければなりません．

それでは実際のプログラムに入りましょう．

例　題

1から100までの整数の2乗を求め，表示しなさい．

考え方

2乗を求める部分を機械語にします．100までですから，ALレジスタを用いた乗算をさせる方法を考えてみましょう．

USR関数ではALレジスタを保存すること

ここで注意しなければならないことは，**USR関数で機械語ルーチンを呼んだときには，ALレジスタの内容を保存しなければならない，**ということです．具体的には解答に示してあります．

2乗する値を100H番地に，結果を102H，103H番地に返します．機械語のルーチンは簡単ですね．

解　答

```
100 CLEAR,&H1B00
110 DEF SEG=&H1B00
120 DEF USR0=0
130 A=100
140  FOR I=1 TO A
150  POKE &H100,I
160  DUMMY=USR0(0)
170  B=PEEK(&H102)+PEEK(&H103)*256
180  PRINT USING"#########";I,B;
190  NEXT I
200 PRINT
210 END
```

結果は逆ワード形式で与えられるので，上位バイトを256倍します．

```
MON↵
hJA0↵
0000 8CCB            MOV     BX,CS
0002 8EDB            MOV     DS,BX
0004 A21001          MOV     [0110],AL
0007 A00001          MOV     AL,[0100]
000A F6E0            MUL     AL
000C A30201          MOV     [0102],AX
000F A01001          MOV     AL,[0110]
0012 CF              IRET
```

ALレジスタの内容を保存するため，110H番地に格納しておきます．

乗算結果を102H番地に格納します．

計算が終了したら，ALレジスタにもどしておきます．

BASICへもどるための命令です．

解　説

プログラムを打ち込み終えたら，BASICにもどり

SAVE "EX52A.BAS" ↵

BSAVE "EX52A.M",0,&H13 ↵

とセーブしてから実行して下さい．プログラム上で特に難しいことはないと思います．ALレジスタを保存することを忘れないで下さい．また，機械語ルーチンの最後はHLTでなく，IRET命令で終了して下さい．これはちょうどBASICのRETURN命令と似た働きをするもので，機械語ルーチンからBASICにもどる場合などに使用します．

ALレジスタの保存はPUSH　A，POP　AXで．

機械語ルーチンはIRET命令で終わる

さて，USR関数の使い方が一応理解いただけたと思いますが，この方法では1回ごとに機械語ルーチンを呼び出しているので，BASICと比べてそれほど早くなりません．そこで，一度にすべてを計算してしまう方法を考えましょう．

練習問題

5－3

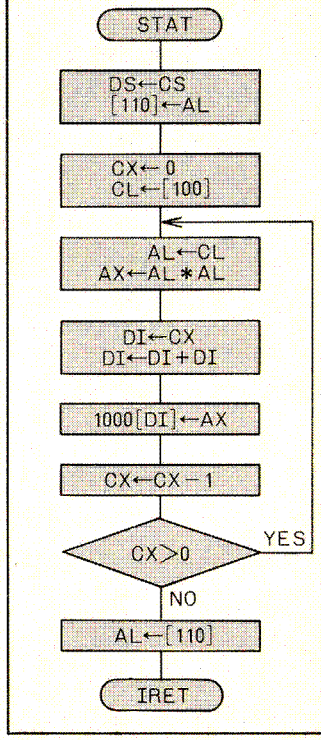
1から100までの整数の2乗を求め，表示しなさい．ただし機械語ルーチンは1回だけ使用し，1度に全部の数の2乗を求めるプログラムを作りなさい．

考え方

この問題を考えるために，基本的な方針を決めておきます．まず第一に，一度に1から100までの2乗を求めるので，結果をメモリ上にストックしておかなければなりません．そのための領域として1000H番地以降を使用します．乗算にALレジスタを用いるとすると結果は2バイトずつになりますから，1000H番地から2バイトずつを使用します．1000H，1001Hは使用せず，1×1の結果を1002H，1003H番地に，2×2の結果を1004H，1005番地に……というように格納していきます．こうすると，アドレスの計算は，1000H＋数値×2で与えられるので楽になります．

被乗数の最大値をCXレジスタに入れておき，減少法で0になったら終了する方法をとります．乗算のためにCXレジスタの下位バイトの値を，ALレジスタに転送します．また，アドレスの計算のために，CXレジスタの値の2倍を計算しなければなりません．そのためのレジスタとして，DIレジスタ（SI，BXなどでもよい）を使用します．それは，これらのレジスタが，アドレスの間接指定に使用できるからです．ここまで決めておけば，プログラムはほぼ作成できるでしょう．

図5.7　フローチャート



解　答

```
100 CLEAR,&H1B00
110 DEF SEG=&H1B00
120 DEF USR0=0
130 A=100
140 POKE &H100,A
150 DUMMY=USR0(0)
160  FOR I=1 TO A
170  A=PEEK(&H1000+I*2)+PEEK(&H1000+I*2+1)*256
180  PRINT USING"########":I,A;
190  NEXT I
200 PRINT
210 END
```

マシン語を使用するプログラムでは，必ずCLEAR文でBASICを使用する上限値を定めておきます．

2乗する数の上限値をメモリの100H番地に格納しておきます．

```
hJL0,20
0000 8CCB          MOV     BX,CS
0002 8EDB          MOV     DS,BX
0004 A21001        MOV     [0110],AL
0007 B90000        MOV     CX,0000
000A 8A0E0001      MOV     CL,[0100]
000E 88C8          MOV     AL,CL
0010 F6E0          MUL     AL
0012 89CF          MOV     DI,CX
0014 01FF          ADD     DI,DI
0016 89850010      MOV     1000[DI],AX
001A 49            DEC     CX
001B 77F1          JA      000E
001D A01001        MOV     AL,[0110]
0020 CF            IRET
```

CXレジスタを，演算数値として使用する場合にはCLのみで使用し，アドレスの計算のためにはCXレジスタとして使用している点に注意して下さい．

2倍のときは乗算命令を使用せず，このようにした方がよい．

解　説

「考え方」のところで充分に説明しましたので，ここでさらに解説する必要はないと思います．言うまでもありませんが，

SAVE "EX15－3. BAS"

BSAVE "EX15－3. M"0, &H21

としてセーブしてから実行して下さい．80文字モードであれば，ちょうど画面いっぱいに出力されるはずです．念のために120行と130行の間に，

125　BLOAD "EX15－3. M"

という行を入れておくと，あとでもう一度実行しようとする際，機械語ルーチンを入れ忘れる，というミスがなくなります．

●素数を求めるプログラム●

2問めに入る前に，素数を求めるプログラムの作成の方法について説明しておきましょう．素数を求める方法には何とおりかありますが，上限の定まっている場合には，「エラトステネスのふるい」という方法を用いるのが最も簡単で早い方法であると言われています．

エラトステネスのふるい

この方法は，最大値をMAXとすると次のようになります．

素数を求める方法

1）2からMAXまでの整数を書き出す．

2）2を素数として，その倍数をすべて消す．(MAXまで)

3）残っている数のうち，今使用した素数を越える数で最も小さい数を素数とする．

4）その素数の倍数を消す．(MAXまで)

5）3）に飛ぶ．(MAXまで)

具体的には次のようになります．

~~②~~ ③ ~~④~~ ⑤ ~~⑥~~ ⑦ ~~⑧~~ ~~⑨~~
~~10~~ ⑪ ~~12~~ ⑬ ~~14~~ ~~15~~ ~~16~~ ⑰ ~~18~~ ⑲
~~20~~ ~~21~~ ~~22~~ ㉓ ~~24~~ ~~25~~ ~~26~~ ~~27~~ ~~28~~ ㉙
~~30~~ ㉛ ~~32~~ ~~33~~ ~~34~~ ~~35~~ ~~36~~ ㊲ ~~38~~ ~~39~~
~~40~~ ㊶ ~~42~~ ㊸ ……

このプログラムをBASICで書くと次のようになります．上表の○／配列を用いて，内容が0か1かで表わします．メモリの制限から，上限を8000とします．

```
100 CLEAR,&H1FFF
110 DEFINT A-Z
120 *MAXIN
130 INPUT"INPUT MAXIMUM (3-8000)";MAX
140 IF MAX<3 OR MAX>8000 THEN *MAXIN
150 DIM A(MAX)
160 FOR I=2 TO MAX:A(I)=1:NEXT I
```

機械語を使用しないので，メモリを十分にとっておきます．

```
170 B=2
180   *LOOP1
190   C=B
200       *CLEARLOOP
210       C=C+B
220       IF C>MAX THEN *NEXTSOSUU
230       A(C)=0
240       GOTO *CLEARLOOP
250   *NEXTSOSUU
260   B=B+1
270   IF B>MAX THEN *PRINTOUT
280   IF A(B)=1 THEN *LOOP1 ELSE *NEXTSOSUU
290 *PRINTOUT
300 FOR I=1 TO MAX
310   IF A(I)=1 THEN PRINT I,
320 NEXT I
330 PRINT
340 END
```

5－4

上記の素数を求めるプログラムの演算部分を機械語に直して実行し，実行時間を比較しなさい．またこのプログラムを改良するとしたら，どんな点があるか考えてみなさい．

解　答

```
100 CLEAR,&H1B00
110 DEF SEG=&H1B00
120 DEF USR0=0
130 BLOAD"EX5-4.M"
140 INPUT"INPUT MAXIMUM";MAX
150 IF MAX<3 OR MAX>8000 THEN 140
160 A1=INT(MAX/256):A2=INT(MAX-A1*256)
170 POKE &H100,A2:POKE &H101,A1
180 DUMMY=USR0(0)
190  FOR I=2 TO MAX
200  IF PEEK(&H1000+I)=&HFF THEN PRINT I,
210  NEXT I
220 END
```

```
0000 50              PUSH    AX
0001 8CC8            MOV     AX,CS
0003 8ED8            MOV     DS,AX
0005 A10001          MOV     AX,[0100]
0008 BE0000          MOV     SI,0000
000B C6840010FF      MOV     BYTE 1000[SI],FF
0010 39C6            CMP     SI,AX
0012 7304            JAE     0018
0014 46              INC     SI
0015 E9F3FF          JMP     000B
```

AXレジスタの内容を，スタックと呼ばれる場所に一時退避させる命令です．詳しくはあとで解説します．

メモリをすべてFFで埋めます．

```
0018 BE0200          MOV     SI,0002
001B 89F3            MOV     BX,SI
001D 01F3            ADD     BX,SI
001F 39C3            CMP     BX,AX
0021 7708            JA      002B
0023 C687001000      MOV     BYTE 1000[BX],00
0028 E9F2FF          JMP     001D
002B 46              INC     SI
002C 39C6            CMP     SI,AX
002E 770A            JA      003A
0030 80BC0010FF      CMP     BYTE 1000[SI],FF
0035 74E4            JE      001B
0037 E9F1FF          JMP     002B
003A 58              POP     AX
003B CF              IRET
```

次の倍数を見つけ，上限値以上なら次の素数を見つけるプログラムにジャンプします．

倍数があれば，00を代入します．

次の素数を見つけるために1を加え，上限値を越えたら終了します．

次の素数が見つかれば，1BH番地へジャンプします．

スタックに退避されていた値をAXレジスタにもどします．

解　説

前述のBASICのプログラムと比べてみると，よく似ていることがわかると思います．さてこのような問題の場合のレジスタの選び方ですが，変数が少ない場合には，基本的にはBASICのプログラムでの変数にレジスタを対応させればいいわけです．ただし，アドレスの間接指定をすることができるレジスタには限りがあることに注意しなければいけません．前述のBASICプログラムで，配列Aの順番を指定するために，変数Bと変数Cが使用されています．そこで変数Bの代わりにSIレジスタを，変数Cの代わりにBXレジスタを使用しています．レジスタはアドレスの間接指定に用いることのできるレジスタなら何でもよいのですが，BPレジスタはセグメント・ベースとして，SS（スタック・セグメント）を使用しているので避けた方が良いでしょう．

配列の添字はポインタ・レジスタに対応させる

A(C)のCはSI, DI, BX, BPのどれかを使用

それぞれ“EX5-4.B”，“EX5-4.M”という名でSAVEした後で実行してみて下さい．機械語ルーチンを使用した場合，8000までの素数は1秒位で求まり，表示が始まります．前述のBASICプログラムでは，8000までの素数を求めるのに8MHzで計算終了まで1分40秒ほどかかります．

改良点ですが，第一に2の倍数すなわち偶数は素数ではないので，あらかじめ抜いておくことにより，速度の向上とメモリの節約ができます．第二に素数であるか否かの判別は0か1かで判別できるので，1バイトを使う必要は無く，1ビットに1つずつ数値を対応させることによりメモリを節約できることなどがあげられます．なお，ビットを1つずつ計算するには，論理演算を用いますが，本書ではこれ以上詳しくはふれません．

3 サブルーチンとスタック

1 サブルーチン

機械後でのプログラムの作り方にも大分なれてきたと思いますので，このへんで機械語プログラムでのサブルーチンの作り方を学びましょう．

サブルーチンとは何か？については，BASICで多少なりともプログラムを作ったり，読んだりした経験のある方にはもうおなじみですが，これは副プログラムとも言われ，一連の処理（演算や入出力など）をブロック化したプログラムのことです．

サブルーチンを使用する利点は，主として次の2つがあげられます．第一に，同じ演算や入出力を行なう部分をサブルーチンとすることで，同じプログラムを何度も書く必要がなくなることです．

第二に，プログラムをいくつかの個別のブロックに分けることで，それぞれのブロック（サブルーチン）の処理内容を明確にすることができ，またブロック別の入れ換えや修正が容易となること，という点があげられます．プログラムになれないうちは，主として第一の目的のためにルーチンを用いることが多いのですが，より高度なプログラムを作るためには，この二点めを重視しなければなりません．

サブルーチンを使う意義

プログラムを作成する場合，サブルーチンを用いて作成することも，用いないで作成することも可能である場合には，サブルーチンを用いてブロック化したプログラムを作成するように心がけるべきです．それは，同じ目的を果たすプログラムであっても，1本のプログラムにだらだらと書かれていると，どこで何をしているのかが不明瞭で，また部分的な修正の必要な場合にもプログラ

ブロック化しよう

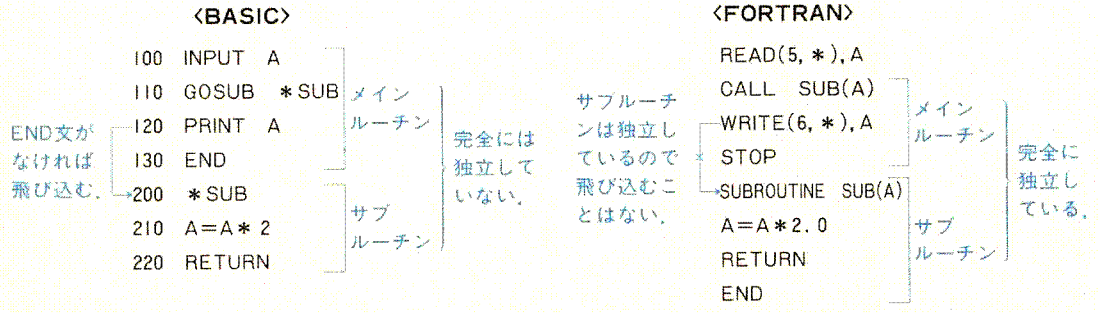
ム全体を作り直さなければならなくなるからです．

それに比べて，サブルーチンを用いてプログラムをブロック化することにより，ひとつひとつの処理がどこでどのように行なわれているのかが容易に理解でき，また修正の必要なときにも，部分的に修正すればよいことになります．しかもサブルーチンとしてすでに作成されたものを引用することができたり，ブロック単位での追加・削除が容易であるなど，利点は数多くあります．

ブロック化のメリット

ところで，サブルーチンと言っても，BASICでプログラムを組む場合には，ひとつのプログラムのなかで連続した行番号の中の一部分を利用して作るわけですから，絶対的にメインルーチンとサブルーチンを切り離しているわけではありません．それにひきかえ，FORTRANやPASCALなどでは，主プログラムと副プログラムとが完全に独立して作成されます．これらの高級言語ではメインルーチンとサブルーチンとを完全に分離して記述する形のものが多いようです．

●メインルーチンとサブルーチン●



それでは，アセンブリでプログラムを作る場合はどうしたらよいのでしょうか．機械語では一本の連続したアドレスを持つメモリ空間にプログラムを組んで行くわけですから，BASICと同じように，どこからどこまでをサブルーチンとするかは，プログラマーが決定しなければなりません．しかもメインルーチンとサブルーチンの間には絶対的な区別があるわけではないので，メインルーチンの終わりをはっきりさせておかなければ，思わずサブルーチンに飛び込んでしまう，といったことも起こり得るわけです（BASICで，メインルーチンの最後にEND文を書き忘れて，サブルーチンに飛び込んでしまった経験をお持ちの方も多いでしょう）．FORTRANのような独立した構造のものでは，そうした間違いは起こりません．

メモリは連続している

サブルーチンが独立して作成できる言語

同じアセンブリでプログラムを作成する場合でも，サブルーチンを独立した形式のものとして作成できるもの（ASM86など）もありますが，本書では内蔵モニタを使用していますので，それらのアセンブリの使用法についてはふれません．

アセンブリや高級言語では，サブルーチンを完全に独立したものとして扱えると言っても，実はどんな言語でも最終的には機械語に翻訳されてから実行さ

れるわけですから，結果としてでき上がったプログラムはやはり一連のメモリ上に置かれるわけで，サブルーチンごとに独立したメモリ空間が存在するわけではありません．ただ高級言語では，それを機械語に翻訳するプログラム（コンパイラ）が，非常にしっかりとメインルーチンとサブルーチンの使用する部分を分けてくれるので，メモリの使い方などを気にせずにプログラムが作成できるのです．

高級言語ではメモリの割り付けはコンパイラが行なう

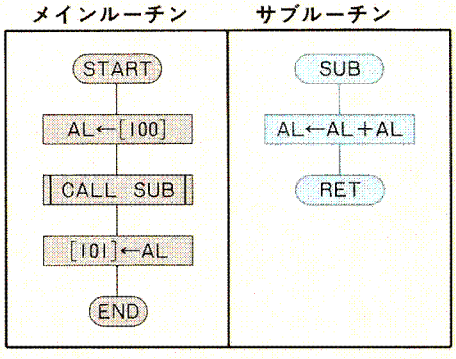
少々横道にそれましたが、例題を一題解いてから，命令をまとめましょう．

例　題

100H番地に格納されている符号なしバイトデータを読み出し，その2倍を，101H番地に格納するプログラムをサブルーチンを用いて書きなさい．ただし結果は，8ビットを越えないものとします．前掲のBASICプログラムを参考にしなさい．

考え方

フロー チャートは右のようになります．サブルーチンの先頭はアドレスで指定しなければなりませんから，ここで“SUB”という名の付けられている部分にアドレスを割り当てる必要があります．ここでは適当に10H位にしておきましょう．サブルーチンの呼び出しにはCALL命令を，サブルーチンからもどるにはRET命令を使用します．



解　答

```
0000 8CC8        MOV     AX,CS
0002 8ED8        MOV     DS,AX
0004 A00001      MOV     AL,[0100]      ← 100H番地からのサブルーチンを呼び出します．
0007 E80600      CALL    0010
000A A20101      MOV     [0101],AL      ← ここは命令が何も存在しない．
000D F4          HLT
000E 0000        ADD     [BX+SI],AL     ← これは関係がない．
0010 00C0        ADD     AL,AL
0012 C3          RET                    ← サブルーチンからもどる．
```

解　説

どうです？別に難しいことはありませんね．問題になるところがあるとすれば，サブルーチンの開始番地でしょう．ここではプログラムの長さを考えて適当に10Hからとしましたが，メインルーチンを入力し終わった段階で，0EH番地以後があいていますからここを先頭に書き始め，あとからCALL命令の飛び先番地を

GOSUBのかわりにCALL
RETURNのかわりにRET

書き換える，という方法もあります．「JMP命令」のところで使った手法ですね．

しかし実際にはメインルーチンとサブルーチンの間は広くあけておいた方が，あとでプログラムを修正するときに楽になる，という利点がありますから，今後は間隔は広くとることにします．実行結果は，各自で確認して下さい．

2 CALL命令とRET命令

それでは，ここでCALL命令とRET命令について解説しましょう．

CALL命令の使い方は，JMP命令とほとんど同じ

2-1　CALL命令

CALL命令は，JMP命令とほとんど同じアドレスの指定法を用います．つまり，直接分岐先のアドレスを指定する方法，レジスタを用いてアドレスを指定する方法，メモリを用いてアドレスを指定する方法，の3通りの方法でアドレスを指定します．さらに，セグメント間のCALLもできます．(ただし内蔵アセンブラは使用できません．)なお，JMP命令にあった，－128d～＋127dに限ったショートジャンプに相当する命令はありません．

●まとめ●

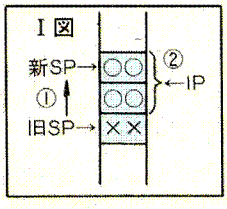
	アドレス指定法	アドレス計算法	分岐範囲	使用法	スタックに退避されるレジスタ及び順序 ＊2
セグメント内	直接(Near-label)	相対	同一セグメント内 ＊1	CALL 1234	IP
	レジスタポインタ間接	絶対	同一セグメント内	CALL AX	IP
	メモリポインタ間接	絶対	同一セグメント内	CALL [BX]	IP
セグメント間	直接(Far-label)	絶対	全メモリ空間	＊3	CS, IP
	メモリポインタ間接	絶対	全メモリ空間	――――	CS,IP

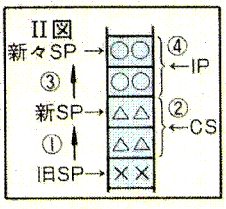
＊1.「相対アドレスの計算法」参照．

＊2.詳しくは「サブルーチンがスタックを使用する理由」で述べる．IPとあるものは，SPの値を2減じたあと，SPの示すアドレスにIPの値（サブルーチンからもどったとき，次に実行されるべき命令が格納されているアドレス）を格納してから目的のアドレスに分岐する．(Ⅰ図参照)

CP,IPとあるものは，SPの値を2減じたあとでまず現在の命令が格納されているコードセグメント（CS）の値を格納し，さらにSPの値を2減じたあとでIPの値（サブルーチンからもどったときに実行されるべき命令が格納されているアドレス）を格納してから目的のアドレスに分岐する．(図Ⅱ参照)

＊3.内蔵モニタのアセンブラでは，この命令は使用できない．別のアセンブラを使用するかまたは，ハンドアセンブルする(実際上使用することはほとんどない．)





2-2　RET命令

RET命令は，サブルーチンの最後に書き，サブルーチンからもとのルーチンへもどるための命令です．RET命令は，どのようなアドレスの指定法でCALLされたサブルーチンであっても，ただ1通りの命令で済みます．

なお，セグメント間コールからもどるためには，RETF命令を使用します．また，ここではまだ使用しませんが，INT命令によって呼び出された割り込み処理ルーチンや，BASICから呼び出された機械語プログラムからもどる場合には，INT命令を使用します．

●まとめ●

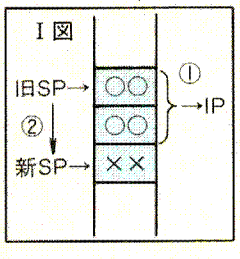
もどる範囲	用　法	スタックから復帰されるレジスタ及び順序＊1
同一セグメント内	RET RET　4　＊1，＊2	IP
セグメント間	――――　＊3	IP，CS
割込み処理ルーチンからの復帰	IRET　＊4	IP，CS，FL

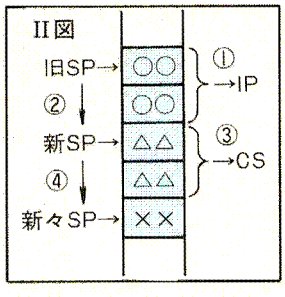
＊1.詳しくは「サブルーチンがスタックを使用する理由」の項で述べる．
IPとあるものは，SPの示すアドレスの内容をIPに転送したうえでSPの値を2減じる（この操作により，もとのプログラムに処理がもどる．）．RET 4などのようにRETのあとに数値の付いた命令を実行すると，さらにSPの値がその数値分だけ減じられる．数値は16進4桁まで可能である．（I図参照）．
IP，CSとあるものは，SPの示すアドレスの内容をIPに転送したうえでSPの値を2減じ，さらにそのアドレスの内容をCSに転送してSPの値を2減じる．この場合にも上記のようにSPの値をさらに減じる命令を作ることができる．（II図参照）
IP，CS，FLとあるものは，上記と同様に処理されたあと，さらにSPの示すアドレスの内容をフラグレジスタに転送してSPの値を2減じる，この命令には，SPの値をさらに減じる命令はない．（III図参照）．

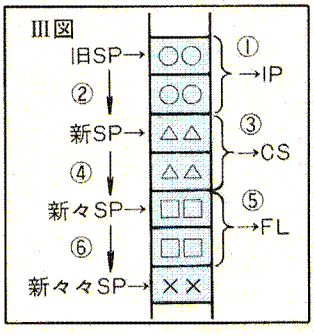
＊2.この命令は、スタックを使用してデータを受け渡すプログラムにおいて使用される．ただし使用法はやや高度なので，本書ではふれない．

＊3.内蔵モニタのアセンブラでは，この命令は使用できない．

＊4.IRET命令については，「割り込み」で説明するので，本節ではこれ以上ふれない．







CALL・RET命令の整理

CALL　即値……即値で示すアドレスからのサブルーチンに分岐します．同一セグメント内への分岐が可能です．

CALL　reg16……レジスタの内容の示すアドレスからのサブルーチンに分岐します．同一セグメント内への分岐が可能です．

CALL　mem16……メモリの内容の示すアドレスからのサブルーチンに分岐します．同一セグメント内への分岐が可能です．

RET……サブルーチンからもとのプログラムにもどります．

RET　数値……サブルーチンからもとのプログラムにもどった後，スタックポインタの値を与えられた数値分だけ増加させます．(数値はもどるアドレスではない点に注意が必要です．)

IRET……割込み処理ルーチンからもとのプログラムにもどります．

※これらの命令ではIRETを除きフラグは変化しない．レジスタの内容は前の表の注にあるもの（IP，CS）を除き変化しない．

※セグメント間コール命令については省略した．

例　題

平面上の2点A(x_1, y_1), B(x_2, y_2)の中点の座標を求めるプログラムを作りなさい．ただし，x_1, y_1, x_2, y_2はそれぞれ1バイトの符号付数値とし，順に100H, 101H, 102H, 103H番地に格納されているものとします．

結果はx座標を104H番地に，y座標を105H番地に格納しなさい．（値は整数で，剰余は無視します．）

考え方

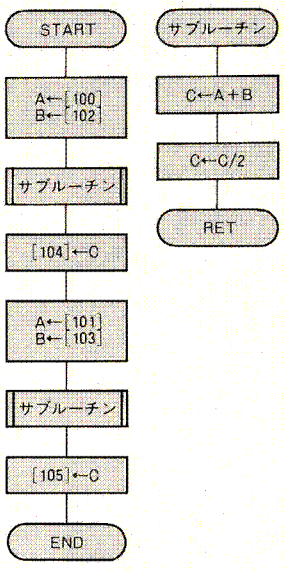
平均値の求め方

$x=\frac{x_1+x_2}{2}$，$y=\frac{y_1+y_2}{2}$　で求められますから，2つのデータをたして2で割るサブルーチンを作り，これを2回使用します．

順を追って考えていきましょう．いきなり，どのレジスタを使用するかを決めるのは難しいので，変数名を用いて考えます．

メインルーチンではまずx座標をA，Bのレジスタに入れてサブルーチンを呼び，結果をCのレジスタに持ちかえってきます。y座標についても同じことをやります．ここから言えることは，A, B, Cの各レジスタはどれも8ビットである，ということです．A，Bの平均がCなので，どれも8ビット以内に納まりますね．

図5.8　メインルーチンとサブルーチン



さて，すると問題はサブルーチンです．(1)を見て下さい．AとBを加えてCとします．このとき，A，Bは符号付き8ビット数値ですから結果は8ビットに納まらないことが考えられます．そこでCのレジスタは16ビットでなければなりません．

ちょっと待って下さい．そうすると(1)の部分は8ビット＋8ビット→16ビットという計算をしなければなりません．ところが8086の加算命令は，必ず加算前と加算後のビット数が一致しなければならないのです．したがって，A，Bは，8ビットの数値を16ビットに変換して加える必要があります．つまり，符号拡張命令を使用する必要があるのです．しかし符号拡張命令はALまたはAXレジスタについてしか行なうことができません．さあ困りましたね．

そこで次のようにメモリの内容をぐるぐる回しにします．A_{16}などとあるのは16ビットレジスタであることを表します．

A_8→ AL
AL→AX　　（符号拡張）
AX→A_{16} ……結果と保存
B_8→ AL
AL→AX　　（符号拡張）

加算時などレジスタ幅に注意

さてここまでで，符号拡張されたデータがA_{16}レジスタとAXレジスタに入っていますから，これを加えて，結果をAXレジスタに格納しましょう．

AX＋A_{16}→AX

したがってA_{16}レジスタとAXレジスタは別のものでなければなりません．そこでA_{16}レジスタをBXレジスタとすればA_8レジスタはBLレジスタになります．そこで残りのB_8レジスタはCLレジスタにでもしておきましょう．すると今までのところは，

BL→AL
AL→AX
AX→BX
CL→AL
AL→AX
AX＋BX→AX　　（代入方向が逆です．注意して下さい．）

ということになります．

次に(2)の部分の割り算です．AXレジスタにはすでに被除数値が入っていますから，これを2で割ればいいのですね．チョット待って下さい！割り算では直

除算は直接数値では行なえない

接数値での割り算はできません．つまり，

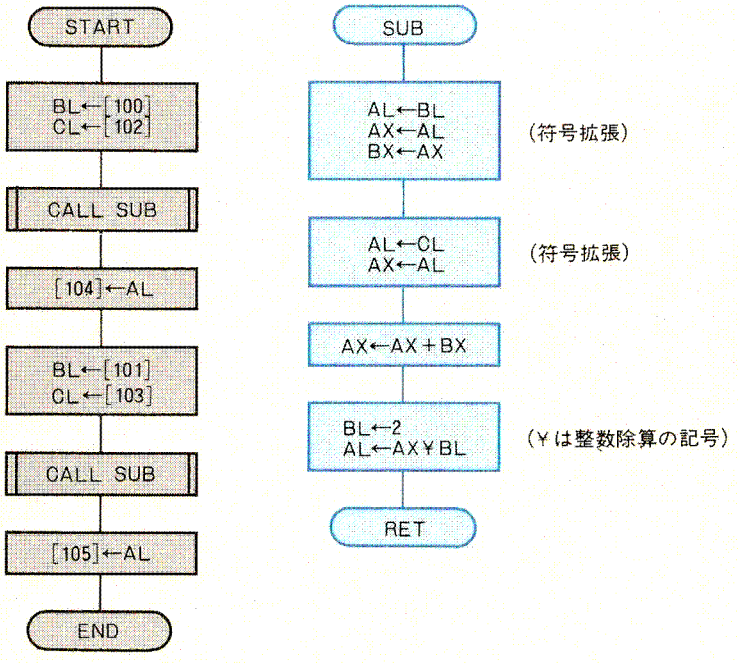
IDIV　　2

は許されないのです．そこで，適当なレジスタに数値2を代入してから除算命令を行ないます．すでにBLレジスタに入っている値は不要ですから，BLレジスタに2を代入して，

IDIV　　BL

を実行すれば，結果の商がALレジスタに，余りがAHレジスタに得られます．余りは無視するので，結果としての商はALレジスタに格納されています．ここまででサブルーチンを終了します．つまりはじめのフローチャートでの復帰情報CはALレジスタを用いることになります．

以上のことから改めてフローチャートにすると，次のようになります．



解　答

これをプログラムにするのは容易ですね．

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 8A1E0001      MOV     BL,[0100]
0008 8A0E0201      MOV     CL,[0102]
000C E82100        CALL    0030  ←――― [30H番地からのサブルーチンを呼びます．
000F A20401        MOV     [0104],AL
0012 8A1E0101      MOV     BL,[0101]
```

```
0016 8A0E0301        MOV     CL,[0103]
001A E81300          CALL    0030
001D A20501          MOV     [0105],AL
0020 F4              HLT

0030 88D8            MOV     AL,BL
0032 98              CBW
0033 89C3            MOV     BX,AX
0035 88C8            MOV     AL,CL
0037 98              CBW
0038 01D8            ADD     AX,BX
003A B302            MOV     BL,02
003C F6FB            IDIV    BL
003E C3              RET
```

Lコマンドで連続して逆アセンブルすると，この命令がずれてしまい，正しく逆アセンブルできません．L30できちんと逆アセンブルし直して下さい．

解　説

説明が長くなりましたが，確実に理解していただけましたでしょうか．

それでは実行してみましょう．とは言っても，データが入力されていないとおもしろくありません．Sコマンドで次のようにデータを入力して下さい．

```
h]S100↵
0100  00－22 00－44 00－66 00－88↵
```

スペースバーを押す

```
h]G0,20
*1B00:0020
h]D100
0100 22 44 66 88 44 E6 00 00 00 00 00 00 00 00 00 00        "Dfl D◥
```

22Hと66Hを加え2で割ると44Hになります．一方，44Hと88Hを加え，2で割ると0E6Hとなるのは，88Hが負の数だから，結果も負となるわけです．

なお，サブルーチンの部分を，わざわざ符号拡張せずに，加算結果がオーバーした場合にはキャリーを用い，ADC命令で16ビットに拡張できないのか，と考えた方もいるかも知れませんが，残念ながら符号付数値は，この方法では加算できません．(オーバーフローフラグ用いるが，難しいのでここではふれない)

オーバーフローフラグは符号付き数値演算に用いる

3 サブルーチンとスタック

どうです，サブルーチンを使ったプログラムの作り方がわかりましたか？サブルーチンを呼び出す時にはCALL命令を，サブルーチンからもとのプログラムにもどる時にはRET命令を使用します．これはBASICのGOSUB命令とRE

TURN命令に相当します．

ところで皆さんは，BASICのプログラムには，なれているでしょうから，GOSUB命令で分岐したサブルーチンからRETURN命令でもとのプログラムにもどることには何の疑問も感じないかも知れませんが，しかしちょっと考えてみて下さい．

RETURN文でなぜもとにもどれるのか

BASICでは行番号の順にプログラムを実行していきます．GOSUB命令に出会うと，指定された行番号にうつってプログラムを実行し，RETURN 命令に出会うと，前に実行中だったプログラムにもどって次の行番号から実行を再開します．おっと待って下さい．前に実行中だったプログラムにもどるためには，前のプログラムの次の行番号がわからなければいけません．いったいどうやって戻る行番号を知ることができるのでしょうか？そうです．どこかに戻る行番号を記憶しておけばよいのです．BASICではGOSUB命令を実行すると，次にもどるべき行番号を自動的にメモリ上の定められた場所に記憶した上で，サブルーチンにジャンプします．そしてサブルーチンの中でRETURNを見つけると，前に記憶していたもどり行番号を読み出して，次に実行する行番号とするわけです．

これと同じことが，機械語のプログラムについても言えます．CPUは機械語をアドレスの順に読み込んで実行していきます．そして，CALL命令を読み込むと，サブルーチンからもどってきた時に次に実行するべきアドレスをメモリ上の，ある特別な領域に記憶した上で，サブルーチンにジャンプします．そしてRET命令を読み込むと，今度はその領域からもどるべきアドレスを読み出し，もとのプログラムにもどるわけです．

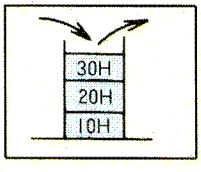
CALL命令はもどり番地を記憶

このように，サブルーチンからもどる番地を記憶するためなどに設けられたメモリ上の領域のことを**スタック領域**または単に**スタック**と呼びます．このスタックに，もどり番地を記憶する方法は，今までのメモリへのデータの格納法と違っています．そこで以下の項で，スタックの働きについて解説をした上で，なぜ，サブルーチンからのもどり番地が，スタックを使用して記憶されるのか，について説明していきましょう．

もどり番地はスタックに記憶

4 スタック動作とスタック・ポインタ

スタックの動作について理解するために，次のような例を考えてみます．1度に1個ずつのデータを持って来て，机の上に積み重ねていきます．順番に，10H，20H，30Hと持ってきたとすれば次の右図のように重ねられています．



次に，このデータを上から順番に持っていきます．すると30H，20H，10Hの順に持っていくことになります．つまり，一番先に持ってきた10Hが一番あとに持っていかれることになりますね．このような「後入れ先出し」動作のことを，**スタック動作**と呼びます．スタック(Stack)とは「積み重ねる」という意味です．

スタックとは積み重ねる動作

メモリの上で，これと同じことを行なっているのがスタック領域です．メモリ上でスタック動作を行なわせるために，**スタック・ポインタ(SP)**というレジスタが使用されます．このレジスタは常に，最後に入力したデータのあるアドレスを示しています．上の例でいえば，3つのデータを持って来た段階で，SPは30Hの存在する場所を示しているわけです．そして，ここからデータを取り出す時には，このSPの示すアドレスからデータを持って行きますので，一番最後に入れたデータを一番最初に取り出すことになります．

スタックポインタは最後のデータのあるアドレスを示す

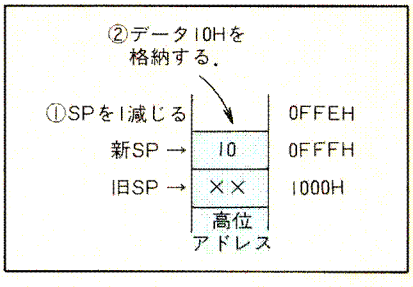
スタックは最後から積み重ねていく

このスタック・ポインタというのは，少々あまのじゃくな動作をし，1つデータを書き込むごとに，高位アドレスから，低位アドレスにむかって動きます．また，1つデータを読み出すごとに，低位アドレスから高位アドレスにむかって動きます．

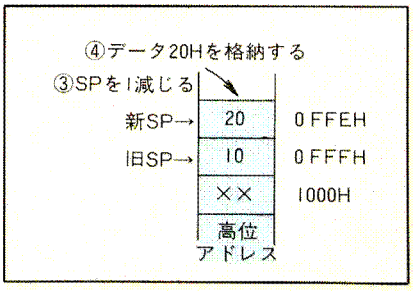
以上のことを具体的に説明するために，前の机上のデータの例が，メモリ上では実際にどう動作するのかを示しておきます．(8086では，スタック動作をするための領域としてスタックセグメントを，コードセグメントやデータセグメントとは別に設けています．したがってこれから述べるスタックポインタのアドレスは，スタックセグメント内でのオフセットアドレスとなります．)

●スタックポインタの動作●

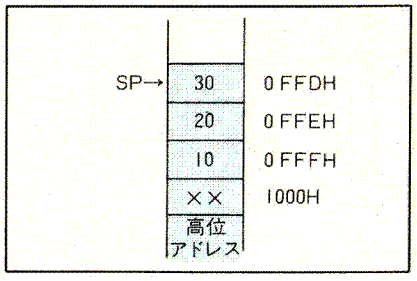
まずはじめにSPの値が1000Hであったとします．そこに，バイトデータ10Hを持って来ると，まず，SPの値を1減じて0FFFHとし，次にこの0FFFH番地に10Hを格納します．



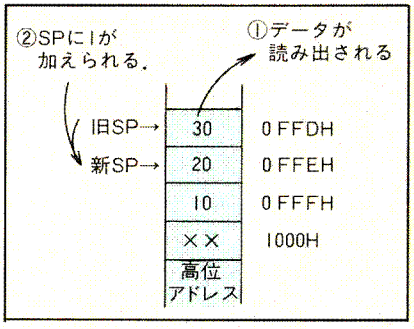
さて，次にバイトデータ20Hを持ってきたとします．するとまたSPの値を1減じて0FFEHとし，次に20Hをそのアドレスに格納します．



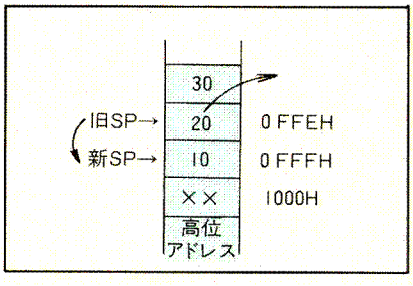
3回目も同様にして，データ30Hを格納し，終わった段階では，SPの値は0DFFDHとなっています．



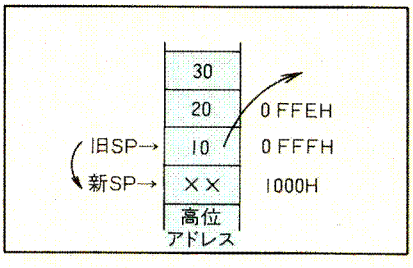
さて，読み出す番です．スタック領域からデータを読み出す命令が実行されると，SPの示すアドレスの内容が読み出されたあとで，SPの値に1が加えられます．この場合には，データ30Hが読み出され，SPの値は0FFEHとなります．



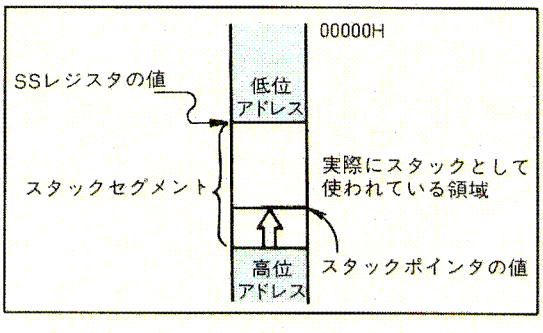
次に読み出す命令があると，再びデータが読み出されSPに1が加えられます．



最後にもう一つデータが読み出されると，一番始めの状態にもどります．ただしこの一連の読み出しの動作では，データは，“読み出される”だけですから，メモリ上には消えずに残っています．しかしSPの値が変わっていますから，スタック上からは消えたことと同じとみて差しつかえありません．



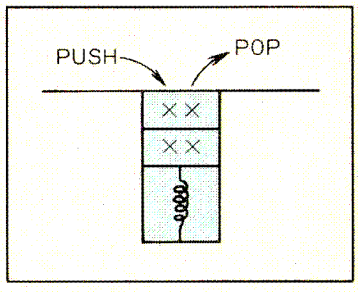
どうです？このようにスタック・ポインタは，常に，最初に取り出すべきデータの格納されているアドレスを示していることがわかりましたね．注意する点は，普通であればメモリにデータを記憶する時に，低位アドレスから順に格納していくところを，スタックを用いた記憶法では，高位アドレスから低位アドレスにむかって格納していく点です。



前にも述べたように，スタックポインタは，スタックセグメントの中のオフセットアドレスを指定します．したがってスタックポインタは，スタックセグメントの高位アドレスがわから始まって，低位アドレスがわにむかって移動し，その間にスタックとしてデータを記憶するわけです．スタックポインタに値を代入することにより，スタックとして使用する区域の初期設定をしたり，設定しなおしたりすることができますが，やたらにこれを行なうとわけがわからなくなるので注意が必要です．通常は，リセット後に初期設定されているので，そのまま使用してかまいません．スタックの便利さは，スタックが今どこを使用しているのかを気にしなくてもデータの出し入れができる，という点にあります（他の記憶領域と重ならないように注意する必要はあります）．

スタック領域は自動的に設定されているので特に気にしなくてよい

5 スタックへの読み書き(PUSH・POP)



それでは実際にスタックにデータを記憶したり，スタックからデータを読み出したりする命令について説明しましょう．

スタックにデータを記憶する命令はPUSH命令，スタックからデータを読み出す命令はPOP命令です．スタックの動作を右図のようなバネのついた箱にたとえると，いかにも押し込んだり(PUSH)，飛び出したり（POP）するというイメージの命令の名前のつけ方ですね．

この命令を使用して，レジスタの値，セグメントレジスタの値，及びメモリの内容をスタックに記憶することができます．

なお，今までの説明では，スタックに1バイトずつ記憶するものとして説明を行なってきました．これは，はじめにスタックの動作を理解するためにはそのほうがわかりやすいと思ったからです．ところが，実際には**8086の命令ではスタックには必ず2バイトずつ書き込み，読み出しを行なう**ことになっています．つまり，スタックに書き込んだり，読み出したりすることができるのは16ビットのレジスタまたは連続した2バイト(1ワード)のメモリ上のデータに限られるわけです．

8086のスタック動作は2バイト単位

具体例をあげてみましょう．

```
PUSH    AX
```

という命令を実行したとします．そうするとまず現在のSPの値を2減じます．次に，新しいSPの示す番地と，その次の番地にAXレジスタの内容を転送します．

このようにスタックは常に2バイト単位で動作します．使用法としては次のような例があります．

```
PUSH    AX
PUSH    CS
POP     1000[BX]
POP     [1234]
```

ただし，**POP　CS　命令だけは存在しません**．CSレジスタに値を代入するためには，IRET命令またはRETF命令を使用します．

PUSH・POP命令の整理

PUSH	reg16	……16ビットレジスタの値をスタックにPUSHする．
PUSH	sreg	……セグメントレジスタの値をスタックにPUSHする．
PUSH	mem16	……メモリ上の16ビットデータをスタックにPUSHする．
POP	reg16	……スタックから16ビットレジスタに値をPOPする．
POP	sreg	(注1)……スタックからセグメントレジスタに値をPOPする．
POP	mem16	……スタックからメモリ上に16ビットデータをPOPする．
PUSHF		(注2)……フラグレジスタの値をスタックにPUSHする．
POPF		(注3)……スタックからフラグレジスタに値をPOPする．

(注1) POP　CSは不可
(注2) フラグレジスタの，使用されていないビットの値は不定のまま転送される．
(注3) フラグレジスタの，使用されていないビットに対して値が転送されても，何も意味を持たない．
※この命令群ではフラグは変化しない．ただしPOPF命令実行後は当然フラグは読み出した値に再設定される．

例　題

①AX，BX，CX，DXレジスタをスタックにPUSHした上で，再び読み出すプログラムを作りなさい．

②CSレジスタの値をDSレジスタに転送するプログラムを作りなさい．

③PUSH，POP命令で特に注意を要するレジスタは何か，全てあげて理由を説明しなさい．

解　答

①

```
0000 50          PUSH    AX
0001 53          PUSH    BX
0002 51          PUSH    CX
0003 52          PUSH    DX
0004 5A          POP     DX      <- (A)
0005 59          POP     CX
0006 5B          POP     BX
0007 58          POP     AX
0008 F4          HLT
```

解　説

スタックの動作は，入れる時と出す時で順番が逆になりますから，上のように，PUSH命令と POP 命令のレジスタの順番を逆にしなければいけません．上のようにPUSH命令を用いてレジスタの内容をスタックに保存した上で，Ⓐの場所に別のプログラムを挿入し，最後にPOP命令でレジスタの内容を呼びもどすようにすれば，Ⓐの部分のプログラムでレジスタの値を変化させても最後にもとの値にもどすことができます．

POSHはレジスタの退避
POPはレジスタの復帰

このような方法は，複雑な計算をする場合，どうしてもレジスタの数が足りなくなってしまう場合などに有効で，よく使用されます．具体的には，さらに学習をすすめていくうちに出てくるでしょう．

このようにレジスタの内容を一時保管することを**レジスタを退避する**といい，取り出すことを**レジスタを復帰する**といいます．以下この用語を使用します．

スタックはデータの一時保管に使用

解　答

②

```
0000 0E              PUSH    CS
0001 1F              POP     DS
0002 F4              HLT
```

解　説

今までプログラムの始めに，データセグメントをコードセグメントに一致させるために，

```
0000 8CC8            MOV     AX,CS
0002 8ED8            MOV     DS,AX
```

という命令を用いてきましたが，ここではスタックを用いて値を転送する方法を示します．

上の例では，まずCSレジスタの内容がスタック上にPUSHされます．次にそのデータをDSレジスタにPOPします．するとCSレジスタの内容がDSレジスタに転送されたことになります．どうです？今までの方法より簡単でしょう．

このように，スタック上にPUSHされたデータは，それがどのレジスタあるいはメモリからPUSHされたのかということとにまったく無関係にＰＯＰすることができます．したがって，上記のような方法により，レジスタなどの内容を転送することができるわけです．

解　答

③

PUSH命令では特に問題はないがPOP命令ではいくつか注意しなければならない命令がある．

POP　　SP　……スタックポインタに値を読み出す．ということは，次のスタック動作が新しく別のアドレスから始められる可能性があることを示している．PUSH　SPで書き込まれたデータを読み出す場合には問題はないが，他のレジスタの値がPUSHされていたものをこの命令で読み出すと，今後のスタック動作がうまくいかなくなることがある．

POP　　SS　……上と同じ理由である．

解　説

別に説明はいらないと思いますが，迷い子にならないためには注意が必要ですね．

練習問題

5－5

100H番地から10FH番地までに，8個のワードデータがある．このデータを，逆順に並び換えるプログラムを，PUSH，POP命令を用いて作りなさい．

解　答

```
0000 0E          PUSH    CS
0001 1F          POP     DS
0002 BB0000      MOV     BX,0000
0005 FFB70001    PUSH    0100[BX]
0009 43          INC     BX
000A 43          INC     BX
000B 83FB10      CMP     BX,0010
000E 72F5        JB      0005
0010 BB0000      MOV     BX,0000
0013 8F870001    POP     0100[BX]
0017 43          INC     BX
0018 43          INC     BX
0019 83FB10      CMP     BX,0010
001C 72F5        JB      0013
001E F4          HLT
```

データセグメントをコードセグメントに一致させます．

100H番地から順に格納する．

転送されるデータはワードデータなので，BXの値を2増加させます．

100H番地から順に読み出す．

解　説

100H番地からのデータを順にPUSHし，再び100H番地以降に順番にPOPすると，何と順番が入れ換わるという魔法のようなことが起こります！

実行に先き立ち，Eコマンドで100H番地からにデータを書き込みます．方法を憶えていますか？

```
hJE100↵
0100 00 11 22 00 00 00 00 00 00 00 00 00 00 00 00 00          "
0110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

画面がクリアされてメモリの内容が表示される．

100H番地以後に00H，11H～FFHまでを書き込みます．

```
0100 00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF          "3DUfwﾌｪｻﾌﾝ/
```

入力し終ったらESCキーを押します．

実行してみましょう．結果は次のようになります．

```
hJG0,1E↵
*1B00:001E
hJD100↵
0100 EE FF CC DD AA BB 88 99 66 77 44 55 22 33 00 11          / ﾌﾝｪｻﾌﾗfwDU"3
```

ワード単位で転送されている．

当然ですが，もう一度実行すると，もとにもどります．

なお，PUSH，POP命令は生成する機械語は短くなりますが，実行時間がMOV命令より若干余計にかかります．両方使用することができる場合には，その点も考慮して下さい．

6 サブルーチンがスタックを使用するのは？

さて，スタックの構造について理解できたでしょうか．それではここでもう一度はじめにもどって，サブルーチンがなぜスタックと関係があるのか，について説明しましょうね．

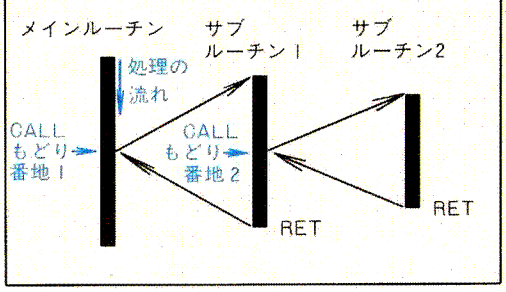
次のような例について考えてみます．メインルーチンからサブルーチン1が呼ばれ，さらにそこからサブルーチン2が呼ばれるような形のプログラムがあるとします．

さて，このプログラムでは，メインルーチン実行中にCALL命令を実行すると，もどり番地1を記憶したうえでサブルーチン1にジャンプします．次に，

サブルーチン1実行中にCALL命令をみつけると，もどり番地2を記憶してサブルーチン2へジャンプします．

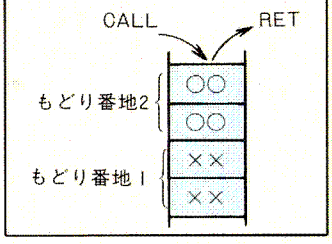
さてここでRET命令に出会いました．もどる場所はどこですか？もどり番地2ですね．このようにRET命令では，あとから記憶したもどり番地2のほうを先に取り出す必要があります．もうおわかりですね．このような目的のためにスタックが，サブルーチンのもどり番地を記憶するのに使われるわけです．

図5.10　プログラムの流れ



多層構造をしたプログラムでは図のようにスタック上にもどり番地がどんどん積み上げられていきます．

図5.11　多構造プログラム

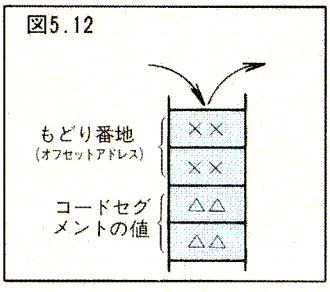


このとき，スタックにPUSHされるのはもどり番地です．すなわちCALL命令実行時のインストラクションポインタ(IP)の値であり，CALL命令の次の命令の格納されているアドレスを指しています．前に一覧表のところで，スタック上にIPの値がPUSHされると言ったのは，こういうことだったのです．例えば，

```
0000 E82D00          CALL    0030
0003 89D8            MOV     AX,BX
```

において，0H番地のCALL命令を読み込んだあとで，IPの値は03Hとなっています．したがってこの0003Hという値をスタックにPUSHしてから，30H番地のサブルーチンに分岐することになります．このへんの動作は10項の「相対アドレスの計算法」に詳しく述べられています．また本節の「命令の整理」の注釈もよく読んでください．

図5.12



セグメント間のサブルーチンコールの場合には，スタックにはもどるべき命令の存在する，コードセグメント(CS)の値と，もどり番地(IP)の値がPUSHされます．

セグメント間コールを行なった場合には，リターン命令もセグメント間リターン命令を使用しなければ，正しくもとのプログラムにもどれないのは，このようにスタック上に積み上げられるデータの数が違うからです．

7 スタックの用途

今までのなかでも，スタックの用途や使い方について，いくらか学んできま

したが，ここで，スタックの用途についてまとめておきましょう．スタックの用途は大きく分けて次の3つがあります．

1．サブルーチンのもどり番地を記憶する．
2．レジスタなどの内容を一時退避する．
3．サブルーチンにデータを引き渡す．

このうち，1と2については一応の使い方を説明してきました．3については，初耳だと思いますが，高級言語のコンパイラなどで関数に値を引き渡す場合によく使われる方法です．レジスタでは引き渡すことのできるデータには限りがありますし，メモリを使用すると，サブルーチンが何個もある場合には，それだけの場所を設定しなければならず面倒です．

そこでそのような場合，データをスタック上にPUSHしてサブルーチンを呼ぶ，という方法をとることがあります．この方法は，大きなプログラムや複雑なプログラムを作る時にはよく使用される方法ですが，ややはん雑で，初心者向きではないので，ここでは解説しません．

上記のように，スタックは多様な目的に使用されるにもかかわらず，ただ1本のひものように連続してしか使用できません．そこで，混乱は起こらないのか，という心配もあるでしょうが，基本的な約束さえ守れば，特に注意しなくても使用できる便利なものです．

その約束とは，PUSHしたものは必ずPOPする，という割とあたり前のことです．特に，**サブルーチンの中ではPUSHとPOPの回数を一致させなければなりません．**もっともわざわざこの約束を無視した，ハイテクニックな技法(3．の方法がこれにあたります)でプログラムを作ることもありますが，今はそんなことは考えずに，安全な方法でプログラムを組んでいきましょう．

例　題

次のようなプログラムを実行したあと，

①どこからがスタック領域として使用されているか．
②スタック領域にどのようなデータが格納されているか．

について示しなさい．

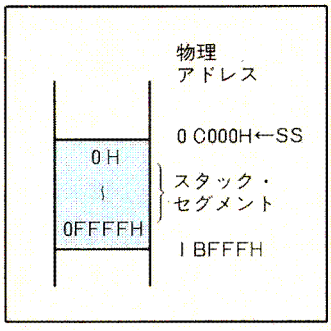
```
0000 B8000C          MOV     AX,0C00
0003 8ED0            MOV     SS,AX
0005 BC0000          MOV     SP,0000
0008 B82211          MOV     AX,1122
000B 50              PUSH    AX
000C 40              INC     AX
000D 50              PUSH    AX
```

```
000E E80100          CALL
0011 90              NOP
0012 50              PUSH
0013 F4              HLT
```

この命令では何も行なわずに次の命令に進めという命令（No Operation）.

解　答

さて，順を追って考えていきましょう．0000H，0003H番地でスタックセグメントに0C00Hを格納します．これでスタックセグメントとして使用されるのは物理アドレスの0C000H～1BFFFH番地（16進5桁）になります．



次にスタックポインタに0Hを設定します．スタックは，設定されたスタックポインタから下位のアドレスに向かって，順に使用されます．それじゃあ始めに0Hに設定してしまったら，全然使えないじゃないか！と思うでしょうがそんなことはありません．

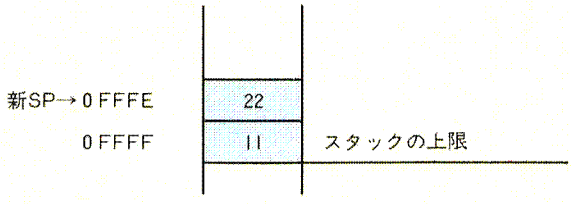
はじめにスタックポインタに0Hを設定しておくと，次にスタックを使用するときにスタックポインタの値が2減じられます．前にも説明したように，8086のスタック動作は必ず2バイト単位で行なわれます．

さて，どうなるでしょうか？0Hから2Hを減じると，0FFFEHとなりますね（借りは無視されます）．したがって，最上位アドレスである0FFFEH，0FFFFH番地にデータが格納されることになります．つまり，始めにSPの値を0Hとすることにより，最上位アドレスから使用することができるわけです（何という逆説的な設定！）

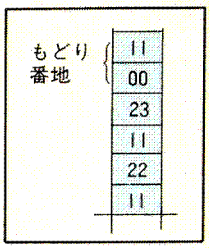
10000H回の繰り返しには0Hを設定

このような方法はくり返しの回数を設定する場合にも使用されます．CXレジスタで繰り返しの回数を減算カウントするようなプログラムでは，はじめにCXの値を0Hにしておくと，10000H回くり返すことになります．

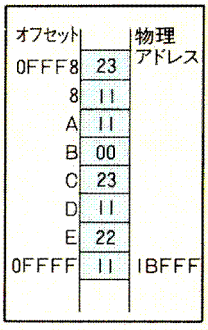
さてそれでは次に進みましょう．000BH番地でAXの内容をPUSHしていますから，これが一番始めにスタック上に格納されます．AXは1122Hですから，逆ワード形式で格納され，下図のようになります．



次にAXの値が1だけ増加され，ふたたびPUSHされます．次にCALL命令で0011H番地にジャンプしますが，その際にもどり番地である0011Hという値をスタックにPUSHします．この時点でのスタック上のデータは右上図のようになっています．

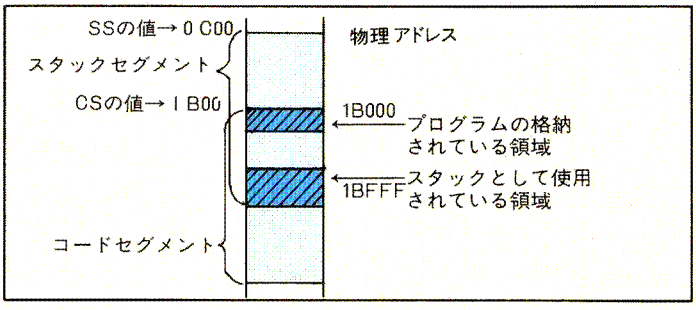


さらにもう1度AXをPUSHします．CALL命令によってAXの値は変化しませんから，また同じ値がPUSHされます．したがって，結果は右下図のようになります．



この問題では，スタックセグメントとして0C00Hをとりましたから，データは絶対アドレスの1BFFFH番地から下位に向かって格納されていきます．このアドレスはいままでコード・セグメントとして用いていた1B00Hを基準にとると，オフセットアドレスが0FFFH番地ということになります．

つまりコードセグメントとスタックセグメントがオーバーラップしているわけで，図で表すと次のようになります．



したがってスタックは，スタックセグメントを基準にすれば0FFFFH番地以下を，コードセグメントを基準にすれば0FFFH番地以下を使用しています．したがって，コードセグメントを基準にしても，スタック上のデータを見ることができます．やってみましょう．

```
hJG0,13↵
*1B00:0013
hJDFF0↵
0FF0 00 1B 14 00 00 1B 02 F0 23 11 11 00 23 11 22 11          ×#    # "
```

この部分が例題のプログラムで使用されたスタック．

このことは，逆に，スタックとプログラムが同じ領域を使用してしまう可能性があることを示しています．下手をするとプログラム自身が書き換えられて，暴走する可能性があるわけです．普通は，そういうことが起こらないように，プログラムやデータの領域と，スタックの領域とが重ならないように，あるいは十分離れて置かれるように設定されます．なお，解答と関係のない部分にも若干データが入っていますが，これは，モニタ自身がスタックを使用しているためです．

第6章

16ビットならではの命令

1　ループ命令

2　ストリング命令

3　割り込みと入出力

1 ループ命令

1 繰り返しの心要なプログラム

5章1節「比較・分岐命令」のはじめのところで，繰り返しの必要なプログラムについて述べました．プログラムにとって繰り返しはなくてはならぬものです．そのために，比較命令や分岐命令が用意されていたわけですね．

それではここでもう一度，例題のプログラムを再録してみましょう．このプログラムは，1dから100d（＝64H）までを加える，というものでした．

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 B96400        MOV     CX,0064
0007 B80000        MOV     AX,0000
000A 01C8          ADD     AX,CX
000C 49            DEC     CX          ┐CXレジスタの内容を1減じ，0でなければ000A
000D 75FB          JNE     000A        ┘番地へジャンプする．
000F A30001        MOV     [0100],AX
0012 F4            HLT
```

繰り返しを簡単にするLOOP命令

この例のように，あるレジスタの内容を1つ減らして，0でなければ繰り返す，という命令は多くのプログラムで使用されています．このようによく使用される命令を，何とか簡単にすることはできないのか，ということで生まれた命令がループ命令です．ループ命令は，上のプログラムの，

```
DEC     CX
JNE     000A
```

という2行をひとまとめにして，

```
LOOP     000A
```

というふうに1行にしてしまおう，という命令です．

つまり，LOOP命令は，CXレジスタの値を1減じて，0でなければ指定のアドレスにジャンプする，という命令です．このジャンプするか否かの判断に使用されるレジスタはCXレジスタ，と決められています．CXレジスタが，「カウンタレジスタ」と言われるのはそのためだったのです．

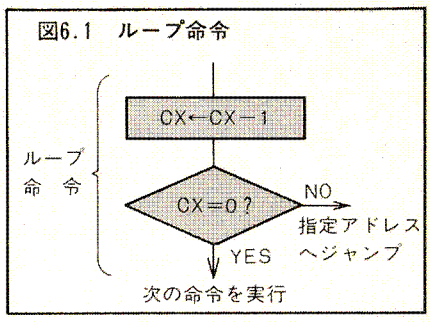


図6.1　ループ命令

カウンタレジスタ

この命令をフローチャートで表わすと，図6.1のようになります．

なお，この命令はCXレジスタの値を1減じて0になったか否かを判断していますが，その際ゼロフラグを含めてすべてのフラグの値は変化しません．したがって，フラブの値は，LOOP命令より前の結果を保持しています．

さて，そこで上記のプログラムをLOOP命令を用いて書き直すと，

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 B96400        MOV     CX,0064
0007 B80000        MOV     AX,0000
000A 01C8          ADD     AX,CX
000C E2FC          LOOP    000A
000E A30001        MOV     [0100],AX
0011 F4            HLT
```

ということになります．若干プログラムが短かくなりましたね．短かくなったといってもたった1バイトじゃないか，と思われるかも知れませんが，このようなこまごまとした節約がトータルでは大きくかかわってくるものです．しかも，LOOP命令の方が，前の2行にわけた命令よりわずかに実行に要する時間が短かくて済みます．このことも，特に繰り返しの多いプログラムでは重要なことになってきます．わざわざこのような命令を作ることにも，それなりの意味があるのですね．

それでは忘れないうちに練習しておきましょう．

例　題

1000H番地以後に00H～0FFHまでのバイトデータを格納するプログラムを，LOOP命令を用いて作れ，

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 BB0000        MOV     BX,0000 ]←―― はじめのデータ＆格納番地．
0007 B90001        MOV     CX,0100 ]←―― 格納するデータの個数．
000A 889F0010      MOV     1000[BX],BL
000E 43            INC     BX
000F E2F9          LOOP    000A ]←―― [CXの値を1減じ，0でなければ000A
0011 F4            HLT                [番地にジャンプする．
```

解　説

練習問題5－1と同じ問題ですが，今度は少し作り方が違います．BXレジスタを，データ格納番地のオフセットとデータそのもの（BLとして使用）として用いる点は共通ですが，データの個数をCXでカウントしています．練習問題5－1では，カウントもBXでやってしまう，という横着をやっていましたが，ここでは個数のカウントには，CXレジスタを専用としました．

転送個数をCXレジスタに

このように，**CXレジスタに，転送するデータの個数をあらかじめ入れておけば，終了条件をあまり深く考えずにデータの転送ができます．** 100H個転送すべきところを，0FFH個しか転送できないという心配は，この方法を用る限り無用です．このようにレジスタに専用の意味を持たせた結果，BXレジスタの値は1回ごとに＋1してやらなければなりませんが，その方がむしろプログラムが何を行なおうとしているのかが，わかりやすくなったと言えるでしょう．

実行結果は，次の通りです．

```
hJG0,11↵
*1B00:0011
hJD1000,10FF↵
1000  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F
1010  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F
1020  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F     !"#$%&'()*+,-./

10E0  E0 E1 E2 E3 E4 E5 E6 E7  E8 E9 EA EB EC ED EE EF     =├┼▲▼♠♥♦♣●○／＼
10F0  F0 F1 F2 F3 F4 F5 F6 F7  F8 F9 FA FB FC FD FE FF     ×円年月日時分秒
```

2 条件判断をともなうループ命令

LOOP命令は，CXレジスタをデクリメント（－1）し，0でなければ繰り返す，というものでした．次に，ただ与えられた回数を繰り返すだけではなく，条件判断も考えあわせて，繰り返すか否かを決定する命令を2つ招介します．

LOOPE 命令は，

CXレジスタを－1した結果が0でなく，
かつ，直前の演算結果が0である（イコールである）

場合にループし，そうでない場合にはループを抜け出す，という命令です．
つまり，CXレジスタの内容がnであった時には，

- n回ループを繰り返す間に演算結果が0でない場合があれば，
ループを抜け出す．
- n回ループを繰り返したら，無条件に，
ループを抜け出す．

ということです．ここでいう結果が0であるか否かは，当然ゼロフラグが1であるか否かで判断しています．したがって「演算」とは，比較命令などでかまいません．上記の表現は，ややわかりにくいので，下記のように憶えて下さい．

イコールになることと**ゼロ**になることとは同義ですから，

LOOPE（LOOP　if　Equal）　イコール（E）であれば繰り返す，（上限CX回）　LOOPE

LOOPZ（LOOP　if　Zero）　ゼロ（Z）であれば繰り返す，（上限CX回）

という2通りの書き方がありますが，同じ命令を別の表現で表しただけです．

もう1つ，混乱しやすい別の命令があります．

LOOPNE（LOOP　if　Not　Equal）

…… イコールでなければ（NE）繰り返す．（上限CX回）　LOOPNE

LOOPNZ（LOOP　if　Not　Zero）

…… ゼロでなければ（NZ）繰り返す．（上限CX回）

という命令ですが，命令中の文章と条件を対応させて考えれば，間違えませんね．

図6.2　LOOPE命令

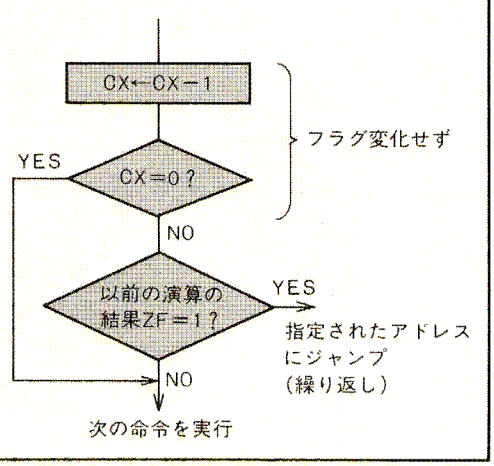
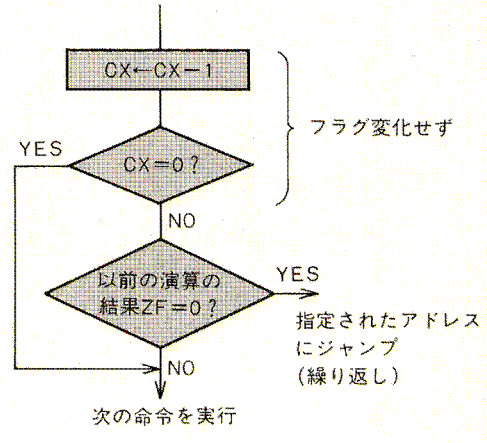


図6.3　LOOPNE(LOOPNE)命令



これらの命令を，フローチャートで示したのが，前ページの図です．これらの命令は他の命令の組み合わせで作ることもできます．この命令の特徴は，はじめにCXの値を1減じて0になるか否かを調べる場合，フラグが変化しない，ということです．したがって，LOOPEやLOOPNE命令は，CXレジスタの値を減じる以前のフラグの状態に従って分岐するか否かを決めています．

それでは各々の命令を使ったプログラムの作り方を練習しておきましょう．

LOOPE命令は0の間中繰り返す命令ですから，**0でないデータをみつける**場合に使うことができます．

次に例題を示します．実はこの例題は不完全なのですが，いきなり完全なものを作ろうとすると複雑になりすぎるので，練習問題をやってから次のステップに進みます．

練習問題

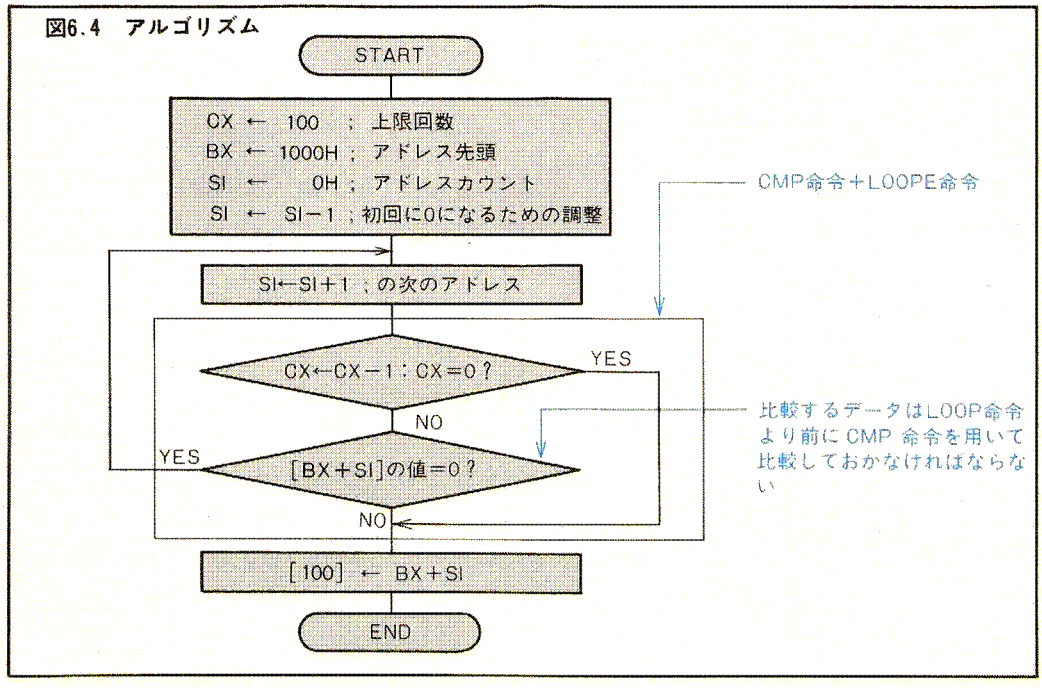
6－1

1000H番地から100H番地以内に0でないデータがあれば，その一番始めのデータのある番地を100H番地に格納せよ．データがない場合の処理は考えなくてよい．

考え方

簡単なアルゴリズムを書いてみます．

図6.4　アルゴリズム



解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 B90001        MOV     CX,0100
0007 BB0010        MOV     BX,1000
000A BE0000        MOV     SI,0000
000D 4E            DEC     SI
000E 46            INC     SI
000F 803800        CMP     BYTE [BX+SI],00
0012 E1FA          LOOPE   000E
0014 01F3          ADD     BX,SI
0016 891E0001      MOV     [0100],BX
001A F4            HLT
```

前ページのワク内は，このように比較命令とLOOP命令の組み合わせで表わされます．

解　説

上のワク内の命令によって，メモリの内容を直接数値０と比較し，イコールで，かつ回数以内であれば次のメモリを調べる，という繰り返しができます．はじめに１回DEC　SIを行なっているのは，１回目のSIの値を０とするためです．

CXに100Ｈを代入しましたから，全部で100Ｈ回までを上限として調べます．したがって，メモリ上の1000Ｈ番地～10FFＨ番地までを調べることができます．

実行する前に，メモリ上にデータを書き込んでおきます．

```
h]F1000,10FF,0↵
h]S1005↵
1005 00-11↵
h]
```

← 1000H～10FFH番地を０にクリア．

← 1005番地に数値11Hを代入する．

それでは実行してみましょう．

```
h]G0,1A
*1B00:001A
h]
```

結果をのぞいてみます．

```
h]D100↵
0100 05 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00
h]
```

逆ワード形式ですが，1005Ｈという値が格納されていますから，正しく探しあてるこそができたわけです．

さて，さきほどこの問題は不完全なものである，と言いましたが，それはどういう点でしょうか？

それを考えるために，LOOPE命令の動作をもう一度考えてみます．

前に示した処理のアルゴリズムをもう一度見てください（図6.3参照）．

最後のデータの比較が行なわれないので注意！

今，例題のように，カウンターの数を100Hにセットします．すると，100H回めにこの命令を実行したときには，CXレジスタの値が0になり，次の命令に移ります．つまり，100H回めには，以前の命令のフラグの状態にかかわらず，ループをぬけ出してしまいます．ということは，100H個めのデータの比較は行なわれなかったのと同じことになってしまいます．

したがって，次のようなことが起こります．

前のプログラムをそのまま使います．

```
h]F1000,10FF,0↵  ←― 1000H～10FFHをクリア．
h]G0,1A↵
*1B00:001A
h]D100↵
0100 FF 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00
h]
```

何を行なったかわかりますか？1000H～10FFH番地までを0でクリアしてからプログラムを実行すると，0以外のデータが発見されないので，最後に調べたアドレスである10FFHを，100H番地に格納して終了しています．

ここで次のようにやってみましす．

```
h]S10FF↵
10FF 00-11↵  ←― 10FFHに11Hを格納．
h]G0,1A↵
*1B00:001A
h]D100↵
0100 FF 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

10FFH番地の内容は0ではありません．しかし100H個目に0以外のデータを発見したか否かにかかわらず，ループ命令は終了してしまいます．そこで100H番地には，上と同じく最終アドレスである10FFHが格納されています．

これでは，最後のアドレスで，0以外のデータが発見されたか否かを判断することができません．困りましたね．

ループ命令の事後処置法

そこで，考えられる方法は2つあります．1つは，繰り返し回数を101H回とする方法です．もう1つは，LOOPE命令終了後も，前の比較命令の結果のフラグレジスタが残っていることを利用して，**データを見つけて終了したのか，100H回の繰り返しが終わって終了したのかを判別する**方法です．

データを見つけて終了した場合には，ZFが0に（等しくなかったということ），データをみつけずに終了した場合にはZFが1になっていることを利用します．

6－2

前問のプログラムを参考にして，1000H～10FFH番地の間に0でないデータがあればその一番はじめのアドレスを，データがなければ0を100H番地に格納するプログラムを作りなさい．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 B90001        MOV     CX,0100
0007 BB0010        MOV     BX,1000
000A BE0000        MOV     SI,0000
000D 4E            DEC     SI
000E 46            INC     SI
000F 803800        CMP     BYTE [BX+SI],00
0012 E1FA          LOOPE   000E
0014 7506          JNE     001C    ←0以外のデータを発見して終了した場合は1CH番地へ．
0016 BB0000        MOV     BX,0000 ←100H回の間にデータが発見できなかった番合は，
0019 E90200        JMP     001E            BXを0にする．
001C 01F3          ADD     BX,SI
001E 891E0001      MOV     [0100],BX
0022 F4            HLT
```

解　説

前半は前のプログラムと同じです．LOOPE命令のあと，0以外のデータを見つけて終了した場合には，JNE命令によって，別の処理ルーチンにジャンプします．0以外のデータを見つけられなかった場合には，BXレジスタに0を代入していますね．

実行結果を見てみます．はじめはすべて0のときです．

```
hJF1000,10FF,0↵  ←――メモリを0クリア．
hJG0,22↵
*1B00:0022
hJD100↵  ←――― 000H番地からは0になっている．
0100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

100H番地が0になっているので見つけられなかったことがわかります．次に，

```
hJS10FF
10FF  00-11↵  ←――― 10FFH番地に数値11Hを代入．
```

```
hJG0,22↵
*1B00:0022
hJD100↵ ←――― 100H番地からは10FFHになっている.
0100 FF 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

次に**LOOPNE命令**を使ったプログラムを作ってみましょう．どのような場合に使用するのか，もうおわかりですね．イコールでない場合にループするのですから，イコールのもの，すなわち**目的のデータを見つけるために**使用します．

6－3

1000H番地から100H番地以内に“41H”というデータがあれば，キャリーフラブを0にクリアし，なければ1にセットするプログラムを作りなさい．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 B241          MOV     DL,41      ←― キャリークラブをクリアする命令です.
0006 B90001        MOV     CX,0100
0009 BB0010        MOV     BX,1000
000C BEFFFF        MOV     SI,FFFF    ←― サーチするデータをDLレジスタに代入しておきます.
000F 46            INC     SI
0010 3810          CMP     [BX+SI],DL
0012 E0FB          LOOPNE  000F
0014 7403          JE      0019       ←― 入力時には，MOV SI,0-1とすればよい.
0016 F9                                   つまりSIに－1を代入したのである.
0017 EB01          JMPS    001A       ←― 目的のデータを見つけて終了した場合には，
0019 F8            CLC                   0019H番地にジャンプします.
001A F4            HLT                ←― キャリークラブをセットする命令です.
```

解　説

初期設定や，ループ部分は前問を参考にすればそれほど難しくないですね．今回は，データをあらかじめレジスタに代入しておきました．また，SIレジスタにもはじめから－1を代入しておきました．

目的のデータを見つけて終了した場合には“イコール”の状態，すなわち，ゼロ・フラグが1の状態でループ命令を抜け出しますから，JE命令でジャンプし，CLC命令によってキャリーフラグをクリアします．そうでない場合には，STC命令によってキャリーフラグが1にセットされます．

キャリークラブの使用法

このように，実行の結果,「うまくいった」か,「うまくいかなかった」かをキャリーフラグにセットする方法は，サブルーチンがメインルーチンにもどる場合に処理がうまくいったか否かを伝える方法としてよく使用されます．メインルーチンでは，キャリーフラグが1になっていたら，「失敗」したものとして，処理を行なうようにしておけばよいのです．

実行してみましょう．まず，データのない時です．

```
hJF1000,10FF,0
hJG0,1A
*1B00:001A
hJX
FL -----S-A-C
AX 1B00
BX 1000
```

どうです？データがみつからなかったのでキャリーフラグがセットされていますね．次に，データを入力してから実行してみましょう．

```
hJS10FF
10FF 00-41
hJG0,1A
*1B00:001A
hJX
FL -----Z-P-
AX 1B00
BX 1000
```

今度は“41H”が見つかったのでキャリーフラグはリセットされていますね．
10FFH番地に格納されていてもOKだったわけです．

LOOP命令の整理

命令		動作（上から順に行われる）
LOOP	アドレス	CX←CX－1（フラグは変化しない） CX≠0ならば指定のアドレスにジャンプする．
LOOPE LOOPZ	アドレス	CX←CX－1（フラグは変化しない） CX≠0でかつZF＝1ならば指定のアドレスにジャンプする．
LOOPNE LOOPNZ	アドレス	CX←CX－1（フラグは変化しない） CX≠0でかつZF＝0ならば指定のアドレスにジャンプする．

※これらの命令ではフラグは変化しない．
※分岐先のアドレスはショートラベルすなわち－128～＋127バイトの範囲内である．したがって，アドレスは直接数値で指定しなければならない（条件分岐命令と同じ）．
※LOOPE/LOOPZ命令，LOOPNE/LOOPNZ命令はそれぞれ同じものを別の表現で表わしたにすぎない．内蔵アセンブラではどちらも使用できるが，逆アセンブルすると，左（上）の形式に統一される．

2 ストリング命令

ストリング命令とは，文字列のように連続したデータの処理を行なう命令です．この命令は単独で使用することもできますが，リピートプリフィックスといわれる命令と共に用いることによって一度に大量のデータを転送したりすることができます．使用法が特殊なので，はじめのうちは奇異に感じられるかも知れませんが，なれてくればこれほど便利なものはない命令です．

リピートプリフィックス

まずはじめに，ストリング命令について次の表で簡単に示しておきましょう．

表6.1 ストリング命令

転送命令群	LODS	メモリからレジスタにデータを転送する．
	STOS	レジスタからメモリにデータを転送する．
	MOVS	メモリからメモリにデータを転送する．
比較命令群	SCAS	レジスタとメモリの内容を比較する．
	CMPS	メモリどうしの内容を比較する．

これらの命令は，バイト単位の操作なのか，ワード単位の操作なのかを表わすために，命令の５文字目にＢまたはＷをつけなければいけません．そこで実際には，

LODSB
STOSW

のように使用します．

1 LODS,STOS命令

さてこれからストリング命令の解説に入ります．ストリング命令では，すべての命令に共通な暗黙のオペランド指定が行なわれますので，その規則を示す

暗黙のオペランド指定

ことにします．

- **メモリからデータを読み出す時にはＤＳ(データセグメント)内のＳＩ（ソースインデックス）で示されるアドレスの内容を読み出す．**
- **メモリにデータを書き込む時にはＥＳ(エクストラセグメント)内のＤＩ(ディスティネーションインデックス)で示されるアドレスに値を格納する．**
- **レジスタを使用する時にはＡＬまたは AX レジスタを用いる．**

したがって，例えば以下の命令では次のようになります．

LODSB ……メモリ上の，データセグメント内の SI で示されるアドレスの内容を AL レジスタにロードする．
（DS：MOV AL, [SI] と同じ）

STOSW ……AX レジスタの内容を，メモリ上のエクストラセグメント内の DI で示されるアドレスにストアする．
（ES：MOV [DI], AX と同じ）

ストリング命令

このような約束があるために，ストリング命令はオペランドを持たない命令となっています．

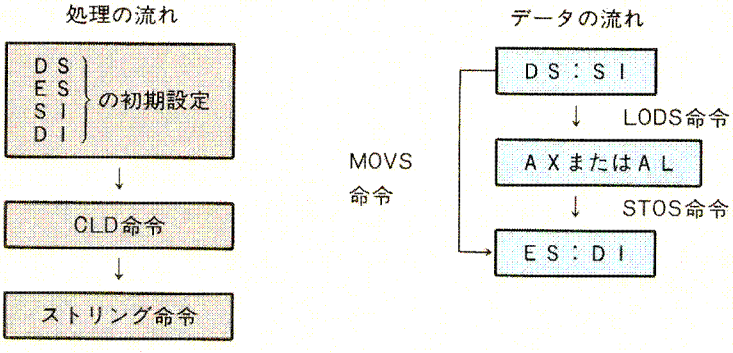
ＳＩあるいはＤＩレジスタの値が自動的に変化

さらにこれらの命令は，実行後，使用したＳＩまたはＤＩレジスタの値が１または２だけ自動的に増加します．（バイト操作命令の場合には１，ワード操作命令の時には２変化します）この機能のおかげで，連続したメモリ上のデータを読み出したり，書き込んだりする場合に，いちいちＳＩあるいはＤＩの値を操作する必要がありません．

ディレクションフラグの値

なお，ディレクションフラグの値が０のとき，レジスタの値が自動的に増加しますが，ディレクションフラグの値を１にすることにより，自動的に減少させることができます．詳しくはあとで述べます．

いろいろなことを一度に説明しましたので，混乱してしまったかも知れませんから，プログラムを組むうえで必要な事項を整理しておきましょう．



という流れになります．

それでは例題に入ってみましょう．

例　題

1000H番地にバイトデータ11Hを，1001H番地にバイトデータ22Hを格納し，LODSB命令，STOSB命令を用いてこれを2000H，2001H番地に転送しなさい．

考え方

前のページの処理の流れ，データの流れを見れば，どんなことをしたらよいか，だいたいわかりますね．繰り返しは使いませんので，フローチャートにする必要はないと思います．

LODSB命令でＡＬレジスタにデータをロードし，STOSB命令でＡＬレジスタからメモリにデータをストアします．この操作により，LODSB命令実行後ＳＩの値が１増加し，STOSB命令実行後ＤＩの値が１増加しますから，続けてLODSB命令，STOSB命令を実行することにより，次のバイトが転送できます．

解　答

```
hJL0,1B
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 8EC0          MOV     ES,AX
0006 BE0010        MOV     SI,1000
0009 BF0020        MOV     DI,2000
000C C606001011    MOV     BYTE [1000],11
0011 C606011022    MOV     BYTE [1001],22
0016 FC            CLD
0017 AC            LODSB
0018 AA            STOSB
0019 AC            LODSB
001A AA            STOSB
001B F4            HLT
```

DSとESを一致させておきます．

SIにロードするデータのソースアドレスをDIに，ストアするデータのディスティネーションアドレスを初期設定しておきます．

ストリング命令の前に，必ずつけておきます．

1000H番地から2000H番地にデータを転送します．

1001H番地から2001H番地にデータを転送します．

解　説

考え方はプログラム中で十分に説明しましたので，もうこれ以上説明する必要はないと思います．プログラムを見るとわかるように，LODS，STOS命令は，オペランドを持たず，生成した機械語も１バイトと短くなっています．このため，初期設定は少々面倒であっても，何度も使用するときには便利です．

実行して，結果をみてみましょう．

```
hJG0,1B↵
*1B00:001B
hJD1000↵
```

```
1000 11 22 00 00 00 00 00 00 00 00 00 00 00 00 00 00
hJD2000
2000 11 22 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

ちゃんと転送されていますね．ついでにレジスタをのぞいておくと，

```
hJX
FL -----Z-P-
AX 1B22
BX 000A
CX 2348
DX 000A
SP F7F6
BP 3C96
SI 1002
DI 2002
CS 1B00
DS 1B00
```

実行後さらにＳＩ，ＤＩの値は１ずつ増加してそれぞれ1002H，2002Hとなっています．
ています．

練習問題

6－4

前問と同じ処理をLODSW，STOSW命令を用いて行ないなさい．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 8EC0          MOV     ES,AX
0006 BE0010        MOV     SI,1000
0009 BF0020        MOV     DI,2000
000C C606001011    MOV     BYTE [1000],11
0011 C606011022    MOV     BYTE [1001],22
0016 FC            CLD
0017 AD            LODSW
0018 AB            STOSW   ←――ワード単位で転送します．
0019 F4            HLT
```

解　説

前の問題では，バイト単位で転送していたのを，ここではワード単位で転送

する方法に変えただけですね．

この命令を実行するとＳＩ，ＤＩの値は１度に２だけ増加します．結果は明らかですから掲げません．各自で確かめて下さい．

2 MOVS命令

この命令も，LODS，STOS命令と基本的な考え方は同じですが，違う点は，メモリからメモリに直接データを転送する命令である，ということです．209ページの表を見るとわかりますが，MOVS命令はレジスタを介さずにデータを転送することができます．

実際にやってみましょう．今までと同じことですから，いきなり練習問題に入っても，大丈夫だと思います．

練習問題

6－5

前問と同じ処理を，MOVSW命令を用いて行ないなさい．

解答

```
0000 8CC8            MOV     AX,CS
0002 8ED8            MOV     DS,AX
0004 8EC0            MOV     ES,AX
0006 BE0010          MOV     SI,1000
0009 BF0020          MOV     DI,2000
000C C606001011      MOV     BYTE [1000],11
0011 C606011022      MOV     BYTE [1001],22
0016 FC              CLD
0017 A5              MOVSW
0018 F4              HLT
```

解説

どうです？ずいぶん簡単になったでしょう．ちょっと実行してみます．

```
hJG0,18↵
*1B00:0018
hJD1000↵
```

```
1000 11 22 00 00 00 00 00 00 00 00 00 00 00 00 00 00
hJD2000↵
2000 11 22 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

ちゃんと転送されていますね．さらに，

```
hJX↵
FL -----Z-P-
AX 1B00
BX 000A
```

としてみると，ＡＸレジスタに1B00Ｈが残っています．つまり，メモリ上のデータを転送する際ＡＸレジスタは使用されず，その前にセグメントの値を転送する際使用した値である1B00Ｈが残っているわけです．

3 リピートプリフィックス命令

さて，これからがストリング命令の本領が発揮されるところです．これまでのところで皆さんは，ストリング命令は，初期設定が面倒な割に，効果の少ない命令だな，と感じられたかも知れません．しかし，これから説明するリピートプリフィックス命令を，ストリング命令とともに用いることにより，ストリング命令は，非常に強力な命令になります．

リピートプリフィックス命令

リピートプリフィックス命令は，その名のとおり繰り返しを指定する前置命令であり，必ずストリング命令の直前に置きます．この命令は，命令自身が何かをする命令ではなく，**ストリング命令の機能拡張を指定する命令である**と考えて下さい．

前置命令

ストリング命令の機能拡張を指定

この命令をストリング命令の前に置くことにより，与えられた回数だけ，ストリング命令が自動的に繰り返されます．

ＭＯＶＳ命令を例にとって説明しましょう．

ＭＯＶＳＢ命令は，ＤＳ内のＳＩの示すアドレスにあるバイトデータを，ＥＳ内のＤＩの示すアドレスに転送した後，ＳＩ，ＤＩの値を１ずつ増加させる命令でした．この命令を１回実行すると１バイト，２回実行すると２バイトの連続したデータが転送できます．ここで，この命令の前にＲＥＰ命令を加えます．すると，ＲＥＰ命令が付いたことにより，次のＭＯＶＳＢ命令は，ＣＸレジスタに指定された回数だけ繰り返されます．具体的な動作としては，ＭＯＶＳＢ命令実行後ＣＸレジスタの値を１減じ，０でなければ再びＭＯＶＳＢ命令を繰り返す，

という動作をします．つまり，ＣＸレジスタに与えられた回数だけMOVSB命令を実行します．

使用法は，

REP

MOVSB

のように使用します．

アルゴリズムを示しておきましょう．

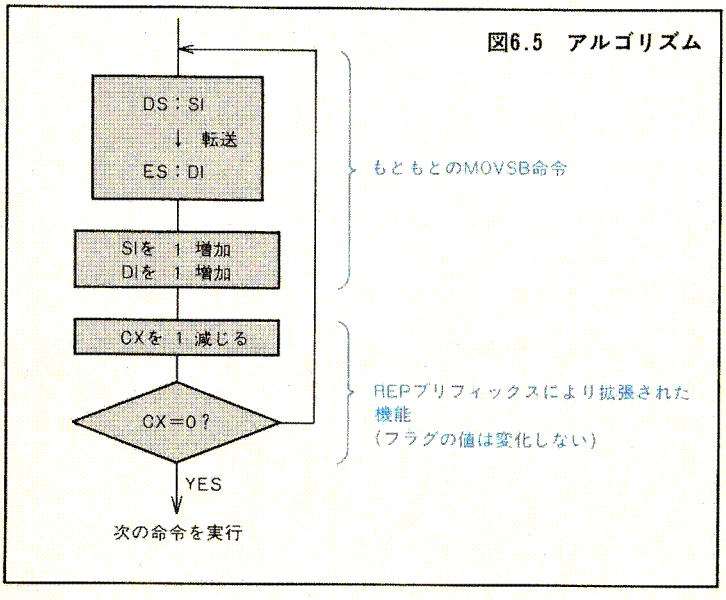


図6.5　アルゴリズム

この命令を実行した結果を，普通のMOVSB命令と比較して図示します．

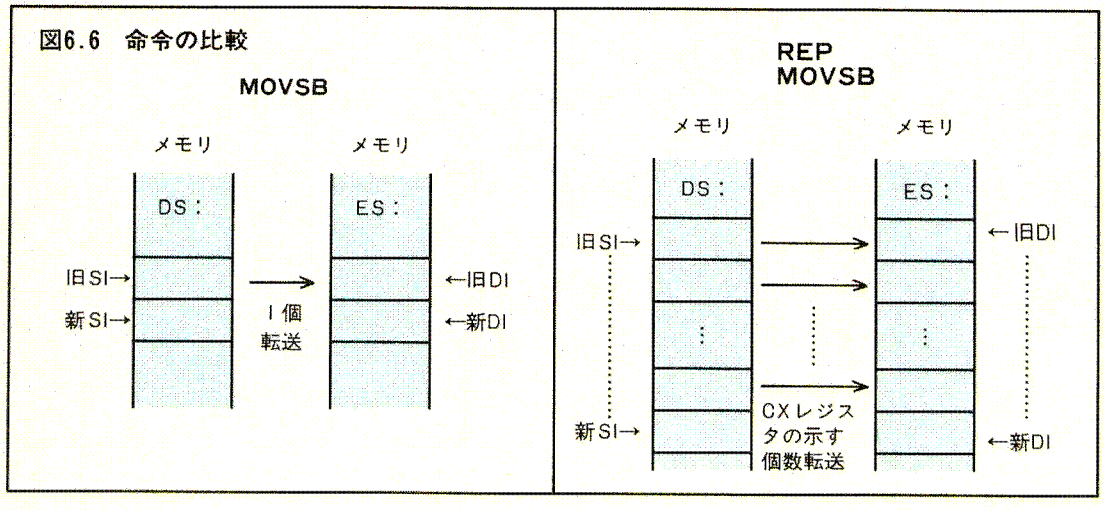


図6.6　命令の比較

練習問題

6－6

データセグメント内の1000Ｈ番地から10FFＨ番地にあるデータを，2000Ｈ番地から20FFＨ番地に転送しなさい．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 8EC0          MOV     ES,AX
0006 BE0010        MOV     SI,1000
0009 BF0020        MOV     DI,2000      ←転送アドレスのセット・繰り返し回数，100H回．
000C B90001        MOV     CX,0100
000F FC            CLD
0010 F3            REP
0011 A4            MOVSB                ←CXレジスタの回数分を転送．
0012 F4            HLT
```

解　説

いままでのストリング命令と同様に，初期設定をします．違う点は，ＣＸレジスタに繰り返す回数をセットしている点です．その後，REPプリフィックス命令とMOVSB命令を続けて書くことにより，MOVSB命令は，自動的に100Ｈ回繰り返されます．

実行する前にデータをセットしておきましょう．少し余分にセットしておきます．

```
h]F1000,11FF,FF
h]G0,12↵
*1B00:0012
h]D2000,210F↵


2000 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
2010 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
2020 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
2030 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF

20F0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
2100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

きちんと100Ｈ個転送されていますね．

このように，REP命令とMOVSB命令を組み合わせることにより100H個のデータを一度に転送することができます．どうです？便利でしょう．

ところで，REP命令でCXレジスタにセットするのは，次のストリング命令の繰り返し回数です．そこで，MOVSB命令のかわりにMOVSW命令を用いると，一度に2バイト（1ワード）ずつ 100回転送することになりますから，全部で200バイトのデータを転送することになります．

ちょっとやってみましょう．前のプログラムがまだ残っているとします．

```
hJA11
0011 A5              MOVSW
0012
```

これでMOVSB命令がMOVSW命令に入れ換わりました．前の問題で，2000H番地から21FFH番地までに0FFHが入っていますからすぐ実行してみます．

```
hJG0,12
*1B00:0012
hJD2000,21FF
2000 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
2010 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
2020 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF

21E0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
21F0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

予定通り200Hバイトのデータが転送されています．もし転送するバイト数が100Hバイトでよければ,CXレジスタの値を50Hにすればよいのですね．

例　題

エクストラセグメントの1000H番地から10FFH番地を，数値0FFHでうめるプログラムを作りなさい．

解　答

```
0000 8CC8            MOV     AX,CS
0002 8EC0            MOV     ES,AX
0004 BF0010          MOV     DI,1000
0007 B90001          MOV     CX,0100
000A B0FF            MOV     AL,FF
000C FC              CLD
000D F3              REP
000E AA              STOSB
000F F4              HLT
```

解　説

STOSB命令を使うと，ＡＬレジスタの値をメモリにストアすることとができますから，これを利用します．ＡＬレジスタに０ＦＦＨを代入しておき，ＲＥＰ命令とSTOSB命令を用いて，メモリ順に書き込んでいきます．ある範囲のメモリを同一の値で埋める時に使用します．

実行すると，

```
h]G0,F
*1B00:000F
h]D1000,10FF
1000 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
1010 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF

10F0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

なお，同じ問題を，STOSW命令を用いて書くこともできます．その場合はＣＸレジスタに50Ｈを，ＡＸレジスタに０ＦＦＦＦＨを代入しておきます．

前述のように，バイト単位の命令のかわりにワード単位の命令を用いて書くこともできますが，これは転送するデータが偶数バイトの場合に限ります．転送するデータが奇数バイトであるかあるいは，データのバイト数があとから決まるような場合にはバイト単位の命令を用いなければなりません．

さて，MOVS命令，STOS命令と述べてきたので，次はLODS命令の解説に入りましょう．

この命令は，REP命令と組み合わせてもあまり意味がありません．

LODS命令は，メモリの内容をＡＬ又はＡＸレジスタにロードする命令ですから，次々と繰り返せば前にロードした値が消されて，新しい値がレジスタに格納されます．したがって，LODS命令の前にREP命令をつけることは，可能ですがあまり意味がありません．

4 転送命令群と比較命令群

それではREP命令の使用法の説明に入る前に，簡単にREP命令とストリング命令の関係をまとめておきましょう．

	ストリング命令	用いるプリフィックス命令	プリフィックス命令の働き
転送命令群	LODS STOS MOVS	REP	ＣＸレジスタにセットされた回数くり返す．
比較命令群	SCAS CMPS	REPE or REPNE	比較の結果がイコール(Ｅ)の間くり返す．(上限ＣＸ回) 比較の結果がイコールでない(ＮＥ)間くり返す．(上限ＣＸ回)．

※転送命令群ではフラグは変化しない．比較命令群では結果に従ってフラグが変化する．

この表からわかるように，ストリング命令の種類によって．用いるプリフィックス命令が違います．

前に説明したLODS，STOS，MOVS命令に対してはREP命令を用い，その働きは，ＣＸレジスタにセットされた回数だけストリング命令を繰り返させることです．

ところが，SCAS，CMPS命令のような比較命令群に対しては，今までのようなREP命令では意味がありません．それは，これらの命令は，レジスタとメモリ，あるいはメモリとメモリの内容を比較し，その結果に従ってフラグをセットするだけで，メモリ上のデータは変化しないからです．したがって，これらの命令は，ただ指定された回数繰り返しただけでは，最後に比較した結果がフラグに残るだけで意味がありません．

そこでこれらのストリング命令に対しては，REP命令は，単にＣＸレジスタの示す回数繰り返すだけでなく，その間に，条件を満たさない場合があれば繰り返しを終了するような働きをするようになっています．

具体的には，REP命令をさらに２種に分割し，REPE命令は，イコールの間繰り返す命令，REPNE命令はイコールでない間繰り返す命令となります．

３－１　REPE命令の使用法

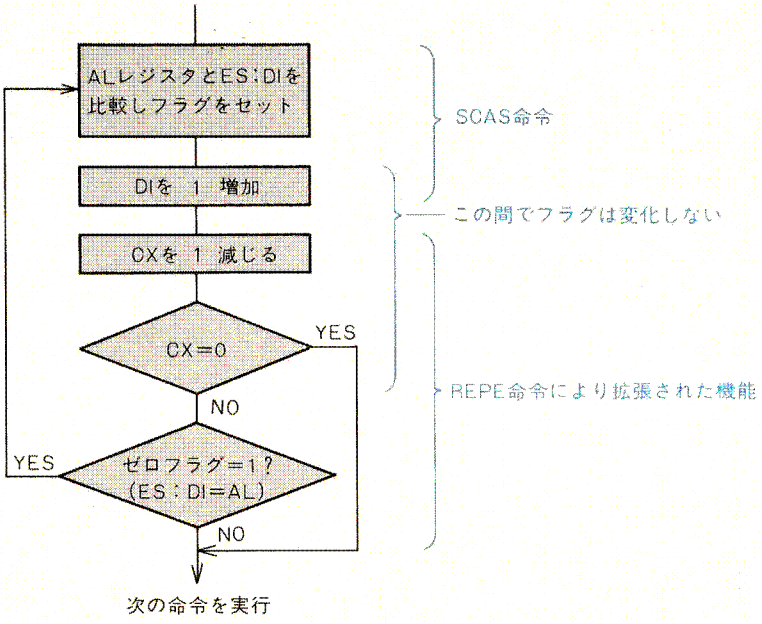
SCAS命令は，メモリの内容とレジスタの内容とを比較する命令ですが，これにREPEプリフィックス命令を前置した場合の，処理の流れを見てみましょう．

右ページのフローチャートを見て下さい．ずいぶん難しそうな命令に見えますが，しっかりと意味をつかんでしまえば，それほど難しいことではありません．

SCAS命令は，比較するだけの命令，REPE命令は，それがイコールの間繰り返す命令です．

したがって，**あるデータ列からＡＬレジスタの値と違うデータを見つけるために使用します．**

ALレジスタの値と異なるデータをみつける



※上のフローチャートはSCASBの場合.
SCASW 命令では，ＡＬレジスタがＡＸレジスタに，ＤＩの増加が２にかわる.

例　題

1000Ｈ番地から 100Ｈ番地以内に０Ｈ以外のデータがあれば，そのアドレスを100Ｈ番地に格納しなさい．見つからない場合には０Ｈを100Ｈ番地に格納しなさい.

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 8EC0          MOV     ES,AX
0006 BF0010        MOV     DI,1000     ←―― データの個数.
0009 B90001        MOV     CX,0100     ←―― 比較するデータとして00Hをセット.
000C B000          MOV     AL,00
000E FC            CLD
```

```
000F F3            REPE    ┐←── イコールの間繰り返す.
0010 AE            SCASB   ┘
0011 7505          JNE     0018←─┐
0013 BF0000        MOV     DI,0000 │ 目的のデータ（0Hではないデータ）を見つ
0016 EB01          JMPS    0019    └ けた場合には分岐する.
0018 4F            DEC     DI
0019 893E0001      MOV     [0100],DI
001D F4            HLT
```

解　説

初期設定については解説の必要はないと思います．REPE命令＋SCASB命令によって，ＡＬレジスタとメモリとを１バイトずつ比較し，イコールの間繰り返します．この間にもしイコールでない場合があれば，繰り返しを終了して次の命令に進みます．この時はイコールでなかったわけですから，次のJNE命令で必ず分岐します．

つまり，必要なデータを見つけて終了した場合にはJNE命令で分岐し，0018H番地のDEC命令に飛びます．ここにDEC命令があるのは，フローチャートを見るとわかるように，SCAS命令が実行されると結果いかんにかかわらずＤＩの値が１つ増加していますから，ＤＩが見つけたアドレスとなるためには１つ値を減じなければならないからです．

この他の処理は前のプログラムとほとんど同じですから，あえて解説を加える必要はないでしょう．

SCAS命令は，ＡＬ（またはＡＸ）レジスタの値とエクストラセグメントのＤＩの示すアドレスの内容を比較する点に注意して下さい．フラグは〔ＡＬ－ＤＩ番地の内容〕の結果によってセットされます．

それでは実行してみましょう．まず，メモリをすべてクリアしておいてから実行して見ます．

```
h]F1000,10FF,0⏎
h]G0,1D⏎
*1B00:001D
h]D100⏎
0100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

1000Ｈ番地からの100Ｈバイト以内に，0Ｈ以外のデータが見つからなかったため，100Ｈ番地には０Ｈが代入されています．

次に，10FFＨ番地に何かデータを入れて実行してみましょう．

```
h]S10FF⏎
10FF 00-11⏎
```

```
hJG0,1D↵
*1B00:001D
hJD100↵
0100 FF 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

今度は，10FFH番地でデータを発見したことがわかります．

なお，この命令は，LOOP命令を用いて書き直すことができます．

```
REPE
SCASB
```

は，

```
L1   ES :
     CMP     AL, [DI]
     INC     DI          (INC命令ではゼロフラグは変化しない)
     LOOPE   L1          (L1はジャンプ先のアドレス)
```

というように書くことができます．

それから，REPE命令はREP命令と同じ機械語コードになりますから，逆アセンブルすると，REPE命令はREP命令に変わっています．これは，REP命令は，転送命令群の前では単なるREP命令として働き，比較命令群の前ではREPE命令として働くことを意味しています．

ここまでくると，REPNE命令の使い方もほぼわかりますね．SCAS命令と組み合わせれば，イコールでない間繰り返すのですから，つまり**イコールのものを見つけ出すのに使います**．基本的には前問と同じですから，自分でやってみましょう．

例　題

エクストラ・セグメント・ベースを0E800Hとし，そこから64Kバイト以内に，0ABCDHというデータ（逆ワードにするとCD，AB）があるか否かを調べ，存在すれば，そのオフセットアドレスをコードセグメント内の100H番地に格納せよ．存在しなければ，0Hを格納せよ．

解　答

```
0000 B800E8          MOV     AX,E800
0003 8EC0            MOV     ES,AX
0005 BF0000          MOV     DI,0000
```

```
0008 B90080          MOV     CX,8000      ←┐ワード単位で64Kバイトさがす場合，
000B B8CDAB          MOV     AX,ABCD       │8000H回カウントすればよい．
000E FC              CLD                   │
000F F2              REPNE                 │
0010 AF              SCASW                 │
0011 7405            JE      0018       ←┐ │イコールすなわち目的のデータが見つ
0013 BF0000          MOV     DI,0000       │かって終了した場合に分岐．
0016 EB02            JMPS    001A          │
0018 4F              DEC     DI          ←┐ │ワード単位のストリング命令では，SIやDIは2ずつ
0019 4F              DEC     DI           │ │増加するので，補正も2減じなければならない．
001A 2E              CS:                  │ │
001B 893E0001        MOV     [0100],DI    │ │
001F F4              HLT                  │ │
```

解　説

ワード単位なので少々手間取ったかも知れませんが，前の問題を参考にすればそれほど困難ではないはずです．セグメントが0E800Hの部分には，N88－BASICのROMが入っています．はたしてROM上に，CD，AB，というデータ列があるでしょうか？

実行してみましょう．ただし以下の結果は，筆者のシステムの場合であって，ROMのバージョンが違う場合には異なった結果となるかも知れません．

```
hJG0,1F↵
*1B00:001F
hJD100↵
0100 00 35 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

オフセットアドレス3500H番地に，目的のデータがあることがわかりました．ちょっとのぞいてみましょう．

```
hJCE800↵
hJD3500↵
3500 CD AB E9 0B 03 E8 BD 01 E8 B7 E6 E9 BB 02 E8 F6
```

目的のデータがちゃんと存在していることが確認できました．

データをメモリ上から探す

このように，REPNE命令は，目的のデータをメモリ上から見つけ出すのに使用されます．皆さんのシステムではどうでしたか？AXに代入する値をいろいろ変えてやってみて下さい．

それでは仕上げにCMPS命令をREP命令と共に用いた問題をやっておきましょう．**この命令は，メモリ上の2つのデータ列が同じか否かを比較する場合に用います．**

練習問題

6－8

1000H番地から5バイトのデータ列と，2000H番地から5バイトのデータ列を比較し，まったく等しい場合にはキャリーフラグを0に，1つでも違う場合にはキャリーフラグを1にするプログラムを作りなさい．

解　答

```
0000 8CC8          MOV     AX,CS
0002 8ED8          MOV     DS,AX
0004 8EC0          MOV     ES,AX
0006 BE0010        MOV     SI,1000
0009 BF0020        MOV     DI,2000
000C B90500        MOV     CX,0005
000F FC            CLD
0010 F3            REP
0011 A6            CMPSB
0012 7403          JE      0017      ←┐比較結果がイコールで終了した場合，キャリーフラ
0014 F9            STC                 ┘グをクリアする命令にジャンプする．
0015 EB01          JMPS    0018
0017 F8            CLC
0018 F4            HLT
```

解　説

CMPSB命令が，メモリ上の2つのデータを比較する命令であることから，イコールの間それを繰り返すREPE（REP）命令を前置すればよいのですね．1つでも違っていると，イコールではない＝ゼロでない，という状態で終了するので，JZ命令ではジャンプせずSTC命令でキャリーフラグがセットされます．

実行してみましょう．

まず1000H番地から，続いて2000H番地からに次のようにデータを入力します．

```
hJS1000↵
1000 00-11 00-22 00-33 00-44 00-55↵
hJS2000
2000 FF-11 FF-22 FF-33 FF-44 FF-11↵
hJG0,18↵
*1B00:0018

hJX↵
FL -------PC
AX 1B00
BX 000A
CX 0000
DX 000A
```

5バイトめが違っていますから，キャリーフラグがセットされていますね．
正しい場合については皆さんで確かめて下さい．

最後に，ストリング命令とリピートプリフィックス命令の整理を掲げておきます．

ストリング命令・リピートプリフィックス命令の整理

ストリング命令		単独動作	リピートプリフィックス命令に前置された場合の動作
			REP
転送命令群	LODSB	DS：SIの示すアドレスの内容をALレジスタに格納後SIの値を1増加させる．	DS：SIの示すアドレスの内容をALレジスタに格納後SIの値を1増加させる．さらにCXレジスの値を1減少させ，0になるまで同じ動作を繰り返す．
転送命令群	STOSB	ALレジスタの内容をES：DIの示すアドレスに格納後，DIの値を1増加させる．	ALレジスタの内容をES：DIの示すアドレスに格納後，DIの値を1増加させるさらにCXレジスタの値を1減少させ，0になるまで同じ動作をくり返す．
転送命令群	MOVSB	DS：SIの示すアドレスの内容をES：DIの示すアドレスに格納後，SI，DIの値をそれぞれ1増加させる．	DS：SIの示すアドレスの内容をES：DIの示すアドレスに格納後，SI，DIの値をそれぞれ1増加させる．さらにCXレジスタの値を1減少させ，0になるまで同じ動作を繰り返す．
			REPE／REPNE
比較命令群	SCASB	ALレジスタの内容と，ES：DIの示すアドレスの内容を比軽し，結果に従ってフラグをセットする．その後DIの値を1増加させる．	ALレジスタの内容と，ES：DIの示すアドレスの内容を比較し、結果に従ってフラグをセットする．その後DIの値を1増加させる．さらにCXレジスタの値を1減少させその結果が0でなく，かつ前の比較の結果がイコール（またはノット・イコール）であれば繰り返す．
比較命令群	CMPSB	DS：SIの示すアドレスの内容とES：DIの示すアドレスの内容を比較し，結果に従ってフラグをセットする．その後SI，DIの値をそれぞれ1増加させる．	DS：SIの示すアドレスの内容とES：DIの示すアドレスの内容を比較し，結果に従ってフラグをセットする．その後SI，DIの値をそれぞれ1増加させる．さらにCXレジスタの値を1減少させ，その結果が0でなく，かつ前の比較の結果がイコール（またはノット・イコール）であれば繰り返す．

注1）　ワード命令の場合，
LODSW，STOSW，MOVSW，SCASW，CMPSW命令の場合，上の動作はすべてワード単位となり，ALレジスタのかわりにAXレジスタが使用されSI，DIの増加量は2となる．

注2）　ディレクションフラグがセットされる場合，
上の動作はCLD命令により，ディレクションフラグが0にクリアされている場合の動作である．STD命令によりディレクションフラグが1にセットされている場合は，SI，DIの値は，指定値だけ減少する．

注3）　これらの命令では，特に指定されている場合を除き，フラグは変化しない．

注4）　比較命令では，前者から後者を減じた結果に従ってフラグがセットされる．

3 割り込みと入出力

1 割り込み処理とは？

皆さんは，BASICのプログラムを走らせとき，間違って無限ループに入ってしまった経験をお持ちでしょう．そんな時，あなたならどうしますか？そう，STOPキーを押せばいいのですね．そうすれば，BASICはプログラムを中断して，コマンドモードにもどってきます．このとき，いったいどんなことが起こっているのでしょうか．

例えば，

```
10 GOTO 10
```

というプログラムを実行したとします．当然無限ループに入ってしまいますね．しかし，機械は書かれたとおりの命令を忠実に実行しているわけです．そこでSTOPキーが押されたとします．すると機械は，プログラムの実行を一時中断して，ROM内のSTOPキーの処理ルーチンに書かれているプログラムに実行を移します．BASICでは，ここには普通プログラムを中断してコマンドモードにもどるような処理が書かれています．

このように，あるプログラムの実行中に，他から別の処理をCPUに要求することを**「割り込み要求」**といい，割り込み要求があった場合の処理法の書かれているプログラムを，**「割り込み処理ルーチン」**といいます．

ところで，このような外部からの割り込み要求(キーボードからの入力は，外部入力である)の有無を知るにはどうしたらよいでしょうか．方法のひとつとして，プログラムの中に常にキーボードからの入力をチェックする部分を入れておく方法です．もう1つの方法は，キーボードからの入力のあった場合にだけ，特別

な信号をCPUに送って割込みをかける方法です．

例えば，あなたが読書をしながら風呂に水を入れる場合を考えてみてください．第一の方法では，あなたは時々風呂の水が一杯になったかどうかをチェックに行かなければなりません．常に時間のことを気にしていなければならず，面倒ですね．第二の方法では，水が一杯になったらブザーが鳴るようにしておき，ブザーが鳴ったら水を止めにいけばよいことになります．こうすればあなたは，ブザーが鳴るまでゆっくり読書を楽しむことができますね．

このように，必要な事態が起こったときにだけ，外部から割り込み要求が起こるようにしておけば，「割り込み要求」がかかるまでの間，CPUはプログラムの実行に専念できることになります．そうでなければ，CPUはプログラムによって常にキーボードやその他の入力装置の状態をチェックしていなければならず，大変効率の悪いことになります．そこで，大抵のマイクロプロセッサでは，外部からの入力を受け付けるために，割り込みを使用しています．8086も当然この割り込み機能を持っています(注)．

注）実際のBASICインタープリタでは，STOPキーの入力は直接に割り込みとして受け付けているわけではないが，ここでは導入のために，たとえとして用いた

2 割り込みの使用法

割込みはこのように，外部の機器からの入力を受け付けるための機能です．ここでその使用法について，例をあげて説明しておきましょう．

前述のSTOPキーの割り込みは，これによって処理が終わってしまったのですが，このような使い方はむしろまれで，一般には，割り込み処理が終了すると，もとのプログラムにもどります．つまり，**割り込み処理は，一種のサブルーチンである，**と考えることができます．

たとえば，何か計算を行ないながら画面に時計を表示する場合を考えてみましょう．PC-9801には，タイマと呼ばれるLSIが内蔵されていて，これを使用することにより，一定時間間隔でCPUに割り込みをかけることができます．そこで，割り込みがかかった場合に時計を表示するプログラムが呼び出されるようにしておけば，計算をしながら時計の表示をすることができます．この場合，計算をするプログラムと，時計を表示するプログラムとは別のものとして作成することができ，CPUも，普段は計算に専念していて，割り込みがかかった時だけ時計を表示すればよく，それが終わればまた計算に専念することができます．

3 割り込み処理と割り込みベクトル

それではここで，割り込みが起こった場合，CPUはどのように動作するのか，前の例を参考にして説明していきましょう．

CPUが計算中に，割り込みが発生したとします．割り込みを起こす装置は，いくつもあります（タイマ，キーボードetc.）から，どの装置から割り込み要求があったのかを知らなくてはなりません．それと同時に，その装置に見合った割り込み処理を行なうためのプログラムが書かれているアドレスを知らなければなりません．

割り込みにはタイプ番号が付く

そこで割り込みには，その種類によってタイプ番号が付けられています．例えば，タイマ割り込みにはタイプ8，キーボード割り込みにはタイプ9というような番号が付けられています．このように，割り込みの起こった装置に従って，割り込みタイプ番号を発生させてCPUに伝える装置として，割り込みコントローラというLSIが使用されています．

さてそこで，タイマからの割り込みがあった場合，CPUはタイプ8の割り込みを受け，それに対する処理先をさがすわけですが，8086ではこのような割り込みの処理先の一覧表を，メモリの物理アドレス0番地からの部分に持っています．1種類の割り込みタイプに対して，セグメントベースとオフセットアドレスが必要なので，1種類に4バイトずつ必要となっています．割り込みのタイプは，タイプ0にはじまってタイプ255まで，全部で256種類ありますから，256×4＝1024バイト，すなわちメモリの先頭から1Kバイトがこの割り込み処理先のアドレスを示すテーブルとして使用されています．このテーブルのことを**割り込みポインタテーブル**または，**割り込みベクトル**といいます．

割り込みは256種類まで可能

ここでその割り込みポインタテーブルを見てみましょう．

テーブルを見ると，タイプ8割り込みに対する処理先は，

セグメントアドレス　　FD80
オフセットアドレス　　195D

であることがわかります．

割り込み処理は，セグメントを越えて分岐

そこでCPUは，タイマ割り込みの処理先として，セグメント0FD80Hの195DH番地からのプログラムに分岐します．その際，通常のサブルーチン・コールと同様に，もどり番地をスタックに記憶します．ただし，普通のサブルーチンと違う点は，割り込み処理においてはセグメントを越えて分岐するため，もどり番地としてセグメントベースとオフセットアドレスの両方を記憶する点です．さらに，フラグの状態も合わせて記憶されます．したがって，サブルーチンの終わりも，RET命令でなく，IRET命令でなければなりません．

なお，この割り込みポインタテーブルは，RAM上に書かれていますので，書き換えることが可能です．上記の場合にはBASICで初期設定されているため，BASIC・ROM上のアドレスになっていますが，ユーザが設定し直すこともできます．256種類の割り込みのうち，いくつかは8086自身が処理のために使用し，また，その他の多くが，PC-9801のBASICの処理用に使用されていますが，未使用の部分もあり，ユーザが自由に設定できるようになっています．

4 内部割り込み

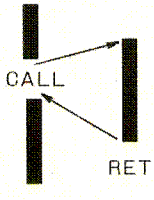
割り込みをサブルーチンとして使用する

さてこのように，割り込みとは，ある処理をしている途中に，外部からの要求によって一時的に別の処理を行なうことであることがわかりました．そして，もどり番地を記憶していて，処理が終わったらもとのプログラムにもどる，という点でサブルーチンと似ているということがわかりました．事実，割り込み処理プログラムは，最後がIRET命令で終わる以外はまったく普通のサブルーチンと変わりありません．

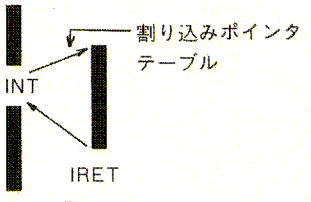
そこで，それを利用して，割り込み処理をサブルーチンのかわりに使ってしまおう，というのが内部割り込み，または**ソフトウェア割り込み**と言われるものです．

ソフトウェア割り込み命令はプログラムの途中に書いておくことにより，外部割り込みが発生した場合と同様に，割り込み処理ルーチンに分岐させる命令です．ソフトウェア割り込み命令はINT（INTerrupt）と書かれます．

それでは，CALL命令の場合と割り込み処理の場合の流れをそれぞれ図に示してみましょう．



CALL命令の場合



割り込みの場合

どうですか？ほとんど同じですね．違う点は，CALL命令が，分岐先を命令の中で指定するのに対して，INT命令は，命令の中に書かれた割り込みタイプ番

号に従って割り込みポインタテーブルから分岐先を読み出す点です．

もう一つ大きな違いはCALL命令が，セグメント内，セグメント間のどちらの分岐もできるのに対し，INT命令は，セグメント間の分岐のみである，という点です．しかし同じセグメント間コール（別のセグメントのサブルーチンへの分岐）でも，INT命令のほうがCALL命令よりも便利です．それは次のような理由によります．

INT命令はCALL命令より便利

第一に，それぞれの処理に対し個別の割り込み番号を決めておけば，いちいち処理先のアドレスを調べることなしに番号のみで分岐先の指定ができる点です．第二に，処理先のアドレスを変更する場合に，割り込みポインタテーブルのみを書き換えれば良い，という点です．同じルーチンを何度も呼び出している場合，CALL命令を使用している場合には何か所も修正しなければならないのに対し，INT命令では，1か所だけ修正すればよいのです．

このような理由から，セグメントをまたがる分岐を行なう際には，ほとんどINT命令が使用されています．さらに，セグメント内コールでもINT命令を使用することがよくあります．これは，上に述べた利点を考えれば当然のことといえるでしょう．

このように，内部割り込みは，本来の割り込み処理とは違い，サブルーチンを呼び出す便利な方法として使用されています．

5 特殊な内部割り込み

内部割り込みには，このようにプログラム中に書かれる割り込み（ソフトウェア割り込み）の他に，CPUが自動的に発生する割り込みがあります．「除算命令」のところで説明した，0による除算の際に発生する割り込み（INT 0）がそれです．この場合，プログラム中にINT 0命令が書かれているわけではありませんが，除算が不可能な場合に，自動的にINT 0命令が実行されることになっています．

また，CPUがシングルステップモードというモードになっている場合には，一命令実行するごとにINT 1割り込みが発生します．これもCPUが**自動的に発生する割り込み命令**です．

このように内部割り込みには，CPUが自動的に発生するものがあります．その他ちょっと変わった命令としてINTO（INTerrupt on Overflow）命令があります．この命令は，オーバーフローフラグがセットされている場合にだけINT 4命令を発生し，それ以外の場合には何もしない命令です．

普通の外部割り込みや，ソフトウェア割り込みでは，タイプ番号を自由に決めることができるのですが，上記の3つの割り込みと，NMIと呼ばれる特殊な外部割り込みだけはタイプ番号が決められていて，変更することはできません．また，ブレークポイント割り込みといわれる割り込みも，通常タイプ3に固定されています．

右に，その固定された割り込みを表に示します．

上記以外の番号の割り込みは，8086を用いたシステムの設計者や，プログラムを作成するユーザが自由に使用することができます．たとえば，PC-9801では，タイマ割り込みはタイプ8となっていますが，他の機種では，別の番号に設定されているかも知れません．

表6-2　固定された割り込み

INT 0	ゼロ・ディバイド割り込み
INT 1	シングル・ステップ割り込み
INT 2	NMI割り込み
INT 3	ブレーク・ポイント割り込み
INT 4	オーバーフロー割り込み

6 割り込みの禁止

割り込みには外部割り込みと内部割り込みがあり，いくつかの特殊な割り込みもあることがわかりました．ところで，CPUは割り込みの要求があったときには，いつでも必ず割り込みの処理をしなければならないのか，というとそんなことはありません．実は，都合の悪いときには，ある種の割り込みを禁止(マスク)することができます．

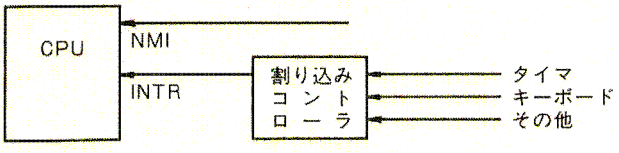
割り込みの禁止

たとえばディスクの読み書きをしている最中に，割り込みがかかってきたとします．データの転送が中途半端のまま割り込みを受け付けてしまうと，処理が終わってもとのプログラムにもどってきても，正しくディスクの読み書きを続けることができなくなってしまいますから，このような場合には割り込みを禁止しておかなければなりません．

このように，割り込みの許可／禁止を決めるために，フラグレジスタの中に，**割り込みイネーブル・フラグ**（IF：Interrupt-enable Flag）というフラグがあります．イネーブル，という名からわかるとおり，このフラグが1のとき，割り込み受け付け可能に，0のときに割り込み禁止になります．このフラグを制御するためにSTI（SeT Interrupt-enable flag）命令と，CLI（CLear Interrupt-enable flag）命令が用意されています．

禁止できるのは通常の外部割り込みだけ

なお，このフラグにより**禁止できる割り込みは通常の外部割り込みだけ**です．通常の外部割り込みは，割り込みコントローラを介してCPUのINTR端子に接続されています．外部割り込みにはもう一つ，**NMI**（Non Maskable Interrupt）というものがあり，こちらのほうは禁止することはできません．またソフトウェア割り込みは，前述のようにサブルーチンの代わりに使用されるものなので，禁止されませんし，CPUが自動的に発生する割り込みも同様に禁止されません．



7 割り込み命令による入出力

それでは例題に入る前に，いったいどのような場合に割り込み命令を使用するのかについて説明しておきましょう．はじめに述べたような外部装置からの割り込みと，その処理ルーチンについては，すでにPC-9801のROMなどに書かれているため，ユーザがわざわざ作成する必要はありません．もっとも，前述の時計表示のような場合は別ですが……．

割り込みはROM内ルーチンコールに用いられる

そこで割り込み命令は，もっぱらサブルーチンをコールするために使用されます．特にBASIC・ROM内のルーチンを呼ぶために使用されます．プログラムを作る人が，自分ですべての入出力ルーチンを作成しなければならないとしたら，大変な苦労がいります．ところが，BASIC・ROMの中には，すでに入出力のためのプログラムや演算のためのプログラムなど様々なプログラムが存在しているのですから，これを利用しない手はありません．そこで，INT命令を用いてこれらのROM内ルーチンを利用すれば，入出力や演算が容易に行なえるわけです．

実は，BASIC・ROM自身も，キーボード入力の部分，ディスプレイ出力の部分，演算の部分，ディスクの入出力の部分等にブロック化されていて，お互いにINT命令を用いて呼び出して使用しています．こうすることにより，プログラムがブロック化され，保守の点からも利用上からも便利になっています．

プログラムのブロック化

それでは，実際に割り込み命令を使用した入力プログラムを作成してみましょう．

例　題

キーボードから1文字入力し，その文字のアスキー・コードを1000H番地に格納しなさい．

考え方

キーボード関係の処理はINT　18Hで行います．INT　18Hの中にさらにいくつもの処理があるため，必要な処理の番号をAHレジスタに格納したうえで，割り込み命令を使用します．この場合は，AHレジスタに0を格納して割り込み命令を実行すると，押されたキーのアスキーコードがALレジスタに格納されてもどってきます．

キーボード入力はINT18H

解　答

```
0000 B400          MOV     AH,00
0002 CD18          INT     18
```

```
0004 2E             CS:
0005 A20010         MOV     [1000],AL
0008 F4             HLT
```

解　説

INT　18H命令は，ROM内ルーチンのキーボードおよびCRT関係の処理を行なうプログラムを呼び出す命令です．このルーチンは，さらにいくつかの部分に分かれているため，コマンドNo.として，AHレジスタに0をセットしてから呼び出さなければなりません．**アスキー・コード**とは，文字や記号をコンピュータで扱うために文字や記号に対してつけられた番号です．これについてはもう御存知ですね．

実行してみましょう．

```
h]G0,8
```

とすると，何も表示されずに待っていますから，ここで何かキーボードを押します．例えば“A”を押したとします．すると，

```
*1B00:0008
h]
```

となって実行を終わります．ここでメモリをのぞいてみましょう．

```
h]D1000
1000 41 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00          A
```

左の数字に対応する文字が表示されている

1000H番地に，“A”に対応するアスキー・コードの41Hが格納されています．また数字列の右側には，数字に対応する文字が表われていますから，41Hが“A”であることがすぐにわかります．このようにして，マシン語でキーボードから入力を受けることができます．

コマンドNo.は毎回セットすること

なお，INT　18H命令実行後，AHレジスタの値は変化してしまいますから，何文字も入力するためには，そのつどAHレジスタに0をセットしてから，INT　18H命令を実行しなければなりません．

例　題

1000H番地からに16文字分のアスキー・コード・データが格納されています．これを画面に表示するプログラムを作りなさい．

考え方

画面出力はINT C4 H

画面に文字を表示するためには，INT　C4H，コマンドNo.3DHを使用します．このコマンドNo.はDIレジスタにセットしなければなりません．このコマンドを使用すると，画面上に1文字だけ表示されます．

なお，INT　C4Hを使用する前に，必ずDS，SSの値を60Hにセットしておかなければなりません．

また，文字を表示した後，改行のために，INT　C4H，コマンドNo.40Hを実行しなければなりません．

解　答

```
0000 B86000          MOV     AX,0060
0003 8ED8            MOV     DS,AX
0005 8ED0            MOV     SS,AX
0007 BE0010          MOV     SI,1000
000A B91000          MOV     CX,0010
000D 2E              CS:
000E AC              LODSB
000F BF3D00          MOV     DI,003D
0012 CDC4            INT     C4
0014 E2F7            LOOP    000D
0016 BF4000          MOV     DI,0040
0019 CDC4            INT     C4
001B F4              HLT
```

使用に先立ってセグメントレジスタの初期設定が必要．

コードセグメント内のSIの示すアドレスから，データをALレジスタにロードする．SIの値が1つ増加する．

画面に1文字表示する．

10H回繰り返す．

改行する．

初期設定がやや面倒ですが，たいしたことはありませんね．LODSB命令や，LOOP命令を使用しましたが動作はわかりますか？DSに60Hを設定しておかなければならないため，LODSB命令の前には，データをコードセグメントから取り出すためのCS：プリフィックス命令が必要となります．

何はともあれ実行してみましょう．準備として，データをセットしておきます．面倒なので，全部同じ文字にしてしまいましょう．

```
h]F1000,100F,42↵
h]G0,1B↵
BBBBBBBBBBBBBBBB
*1B00:001B
h]
```

うまくいきましたね．プログラムの最後の改行命令がないとどんなことになるのか，興味のある人はためして下さい．このプログラムのまま，

```
h]G0,16↵
```

とすればいいのです．変なことが起こったでしょう？

このように，INT命令には様々の種類のものが用意されているので，これら

を有効に利用することができれば，非常に便利です．最後に仕上げの練習問題をやっておきましょう．

練習問題

6－7

キーボードから入力を受け，それが英大文字であれば英小文字に変換して表示しなさい．その他の文字はそのまま表示しなさい．リターンキーが入力されたら終了するものとします．

考え方

英文字のアスキーコードは次のようになっています．(数値はすべて16進数)

大文字を小文字に変換

A	B	C	D	……	Z
41	42	43	44	……	5A
a	b	c	d	……	z
61	62	63	64	……	7A

したがって大文字を小文字に変換するには20Hを加えればよいことがわかります．リターンキーのアスキー・コードは0DHです．

解　答

```
0000 B86000          MOV     AX,0060
0003 8ED8            MOV     DS,AX
0005 8ED0            MOV     SS,AX
0007 B400            MOV     AH,00
0009 CD18            INT     18
000B 3C0D            CMP     AL,0D      ←― リターンキーなら終わる．
000D 7411            JE      0020
000F 3C41            CMP     AL,41      ←― キー・コードが英大文字の範囲外であれば，
0011 7206            JB      0019           そのまま出力へジャンプ．
0013 3C5A            CMP     AL,5A
0015 7702            JA      0019
0017 0420            ADD     AL,20      ←― 英大文字から英小文字に変換．
0019 BF3D00          MOV     DI,003D
001C CDC4            INT     C4
001E EBE7            JMPS    0007       ←― 繰り返し．
0020 BF4000          MOV     DI,0040
0023 CDC4            INT     C4
0025 F4              HLT
```

解　説

例題の2つのプログラムを参考に作成すれば，それほど難しくないですね．英大文字のみを選んで，英小文字に変換するために，アスキー・コードの範囲でふるい分けを行ない，20Hを加えています．

実行してみましょう．

```
h]G0,25⏎
```

ここで何かキーボードを押すと，その文字が表示されます．英大文字は英小文字に変換され，その他の文字はそのまま表示されます．終了にはリターンキーを押して下さい．

```
abcdef12(?⏎            ABCdef12(?
*1B00:0025
h]
```

ABCdef12(?と入力した場合．

INT命令の整理

INT　数値(番号)………割り込みポインタテーブルを参照して，番号の示す割り込み処理ルーチンに分岐する．

INTO………オーバーフローフラグが1にセットされている場合だけ，INT 4割り込みを発生する．

IRET………割り込み処理ルーチンからもとのプログラムにもどる．

STI………割り込みイネーブルフラグを1にセットし，外部割り込みの受け付けを可能にする．

CLI………割り込みイネーブルフラグを0にリセットし，外部割り込みを禁止する．

第7章

さらに知識を深めるために

1　論理演算命令と
　シフト命令

2　I／O命令

3　その他の命令

1 論理演算命令とシフト命令

ここまで，練習問題などをまじえながら，マシン語の使い方を学んできたわけですが，十分に理解していただけたでしょうか．ここから先は，やや使用法の難しい命令について解説していきます．したがって，自分で使用することはできなくても，他の人の作成したプログラムを見て理解することができるようになればよいと思います．説明も，今までのように，1命令にひとつずつ問題練習を行なうようなことはせず，簡単な例を掲げるに留めましたが，使用法の一覧表だけは，きちんとしたものになっていますから，今後の学習に役立てて下さい．

1 論理演算命令

論理演算とは，算術演算のように数値の計算を行なうのでなく，真か偽かによって結果を得る演算です．コンピュータでは，1を真，0を偽とみて，1と0との演算を行なうのが論理演算です．論理演算には**AND**（論理積），**OR**（論理和），**EXclusive OR**（排他的論理和），および**NOT**（否定）の4種類があります．AND，OR，XORでは，2つの数値から1つの結果を得，NOTでは1つの数値から1つの結果を得ます．入力に対する結果を次の表にまとめます．

AND （論理積）
OR （論理和）
EXclusive OR （排他的論理和）
NOT （否定）

入力1	入力2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

入　力	NOT
0	1
1	0

論理演算はこのような命令ですから，基本的に1ビットずつの計算を行なうことになります．そこで8ビットの数値どうしの論理演算を行なうと，1度に8個の論理演算を同時に行なうことになります．

論理演算はビット単位の演算

```
データ1　　10110111　　＝0B7H
　　　　　&&&&&&&&
データ2　　10001101　　＝　8DH
　　　　　↓↓↓↓↓↓↓↓
結果　　　10000101　　＝　86H
```

上図のように，2つのデータからAND命令により結果を得ることを，２つのデータのANDをとる，といいます．見てわかるとおり，各ビット単位でANDがとられて結果が求まっています．両方ともに1になっているビットだけ，結果も1になります．

ANDをとる

実際に使用する場合には，たとえばＡＬレジスタに0B7Hを代入しておいて，

```
AND　　AL, 8DH
```

とすると，結果の86HがＡＬレジスタに格納されます．ＯＲ，ＸＯＲについても用法は同じです．

それでは論理演算命令の具体的な使用法について説明していきます．

1－1　NOT命令の使用法

NOTは反転

NOTは１つ１つのビットを「反転」させる命令です．たとえば，

●8ビットデータのNOTをとった場合

```
データ　　10110111　　＝0B7H
　　　　　NOT
　　　　　⇓
結果　　　01001000　　＝　48H
```

というふうになります．使用法は，

```
NOT　　AL
```

とすることによって，ALレジスタのNOTをとった値が再びALレジスタに格納されます．

1－2　AND命令の使用法

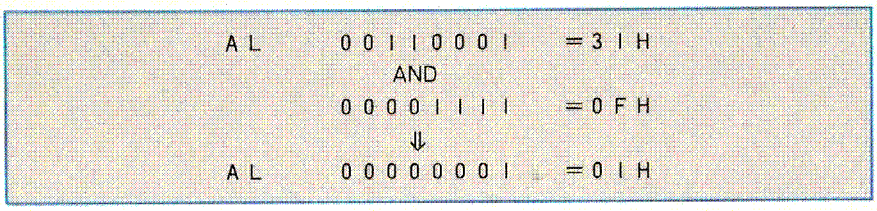
ＡＮＤ命令はビットのマスク

まずAND命令ですが，AND命令では，どちらか一方でも０があればそのビットは０になりますから，これを利用して，**ビットのマスク**に使用します．

たとえば，0FH（＝00001111Ｂ）は，上４ビットが０ですから，どんな数値でも，これとANDをとれば上４ビットは０になってしまいます．キーボードから入力された数値は，０～９がアスキー・コードの30Ｈ～39Ｈに対応しています．そこで，このアスキーコードを数値に直すためには，

```
AND　　AL, 0FH
```

とすればよいのです．ＡＬレジスタにアスキー・コードを入れておくと，数値になって格納されます．



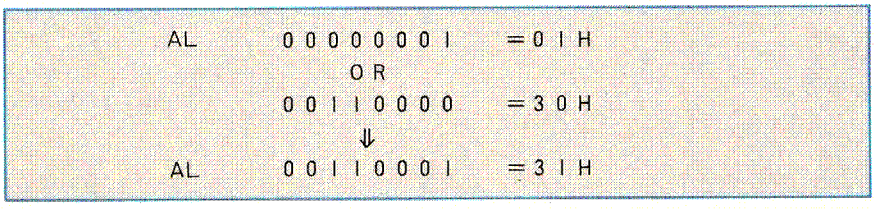
1－3　OR命令の使用法

AND命令とは逆に，OR命令ではどちらか一方でも1があれば，そのビットは1になります．これを用いて，任意のビットを1にすることができます．

OR命令は任意のビットを1にセット

さきほどとは逆に，数値の0～9を，アスキー・コードに直して出力するための方法を考えてみましょう．

```
OR      AL, 30H
```



1－4　XOR命令の使用法

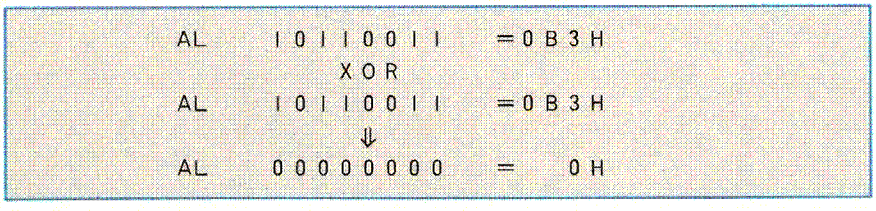
XOR命令は，どちらか一方のみが1のときだけそのビットが1になる命令です．したがって，両方とも0または両方とも1のときには，そのビットは0になります．

この命令も様々な用途に使用されますが，よく使用されるのは次のような使い方です．

XORはビットの反転

```
XOR     AL, AL
```

何をする命令かおわかりですか？ ALレジスタどうしのXORをとっています．同じ値どうしのXORをとるわけです．どんなことが起こるでしょうか？



上の図からわかるように，どのビットも同じものどうしですから，1と1で0になるか0と0で0になり，結局すべてのビットが0になります．そうです，この命令はALレジスタを0にする命令なのです．この命令のほうが，

XOR命令はクリアによく利用される

```
MOV     AL, 00
```

という命令よりも，バイト数も短く，実行時間も早くなります．

1－5　NEG命令の使用法

NEG命令は，2の補数を求める命令です．つまり補数表示を用いた数の**正負を反転させる命令**です．例えば，8ビットの符号付数値1を－1に変換する場合には次のようになります．

NEG命令は正負を反転

```
AL    00000001    ＝01H  ＝01H
        NEG
          ⇓
AL    11111111    ＝FFH＝－01H
```

この命令は，単純なビット操作ではありませんが，他の命令の組み合わせで表現すれば，NOTをとった後で1を加える，という操作で実現できます．

```
NEG     AL
```

は

```
NOT     AL
INC     AL
```

と同じ結果になります．

1－6　TEST命令の使用法，

TEST命令は，基本的にAND命令と同じ働きをしますが，違う点は，実行の結果に従ってフラグを変化させるだけで，レジスタ等の内容を変化させない点です．もとのデータを変化させずに，特定のビットの状態を調べる場合に使用します．

TEST命令はAND命令に同じ

たとえば

```
TEST    AL, 80H
```

とすることにより，ALレジスタの最上位ビットが1かどうかを調べられます．

```
AL    10001010    ＝8AH
        TEST
      10000000    ＝80H
          ⇓
AL    10000000    ＝80H……結果は格納されない
 フラグのみ変化
```

最上位ビットが0であれば結果は0になりますから，ゼロ・フラグがセットされます．また最上位ビットが1であれば結果は0ではありませんから，ゼロ・フラグはリセットされます．したがって，最上位ビットの状態を，ゼロ・フラグの状態で知ることができるわけです．この方法で，もとのデータを変えるこ

ビットの状態を調べる

となく，データの正負を調べることができます．通常，TEST命令のあとには分岐命令で，分岐を行ないます．

論理演算命令の整理

AND	reg, 即値	……レジスタと即値とのANDをとり，結果をレジスタに格納する
	mem, 即値	*……メモリと即値とのANDをとり，結果をメモリに格納する
	reg, reg	……レジスタとレジスタとのANDをとり，結果をレジスタに格納する
	mem, reg	……メモリとレジスタのANDをとり，結果をメモリに格納する
	reg, mem	……レジスタとメモリとのANDをとり，結果をレジスタに格納する
OR	reg, 即値	……レジスタと即値とのＯＲをとり結果をレジスタに格納する
	mem, 即値	*……メモリと即値とのＯＲをとり結果をメモリに格納する
	reg, reg	……レジスタとレジスタとのＯＲをとり結果をレジスタに格納する
	mem, reg	……メモリとレジスタとのＯＲをとり結果をメモリに格納する
	reg, mem	……レジスタとメモリとのＯＲをとり結果をレジスタに格納する
XOR	reg, 即値	……レジスタと即値とのXORをとり結果をレジスタに格納する
	mem, 即値	*……メモリと即値とのXORをとり結果をメモリに格納する
	reg, reg	……レジスタとレジスタとのXORをとり結果をレジスタに格納する
	mem, reg	……メモリとレジスタとのXORをとり結果をメモリに格納する
	reg, mem	……レジスタとメモリとのXORをとり結果をレジスタに格納する
NOT	reg	……レジスタのNOTをとり結果をレジスタに格納する
	mem *	……メモリのNOTをとり結果をメモリに格納する
TEST	reg, 即値	……レジスタと即値とのANDをとり結果に従ってフラグを変化させる
	mem, 即値	*……メモリと即値とのANDをとり結果に従ってフラグを変化させる
	reg, reg	……レジスタとレジスタとのANDをとり結果に従ってフラグを変化させる
	mem, reg	……メモリとレジスタとのANDをとり結果に従ってフラグを変化させる
NEG	reg	……レジスタの内容の２の補数を求め結果をレジスタに格納する
	mem *	……メモリの内容の２の補数を求め結果をメモリに格納する

※これらの命令群では結果に従って，フラグがセットまたはリセットされる

例　題

トラップフラグを反転させるプログラムを作りなさい．

```
0000 9C              PUSHF
0001 58              POP     AX
0002 350001          XOR     AX,0100
0005 50              PUSH    AX
0006 9D              POPF
0007 F4              HLT
```

まず，フラグレジスタはそのままでは，比較も演算もできませんから，ＡＸレジスタに転送するために，スタックを用いて転送を行ないます．フラグレジスタを転送する命令として，LAHF，SAHF命令がありますが，この命令ではフラグレジスタの下位８ビットしか転送できません．そこで，フラグレジスタをすべて転送するには，上のようにしなければいけません．

さて，フラグの配置は，

ビット	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					O	D	I	T	S	Z		A		P		C

のようになっていますから，ビット８がトラップフラグです．そこでAXレジスタのビット８の状態を調べるために，AXレジスタと0100（＝0000000100000000B）とのXORをとります．XOR命令を使用すると，１のビットの部分に重なるビットだけが反転します．０とのXORをとっても，そのビットは変化しません．（表をよく見て下さい．）そこでこの場合，ビット８のみが反転するわけです．

このようにXOR命令は，任意のビットを反転させるために使われます．問題が，トラップフラグをクリアせよ，という問題であれば，XORのかわりに**AND　AX, 0FEFFH**とします．こうすると０となっているビット８のみがクリアされます．

また，トラップフラグをセットする問題であれば，ここにはＯＲ命令を用いて**OR　AX, 0100H**とします．ＯＲ命令では，１になっているビットのみが新たにセットされます．以上をまとめると，

AND命令は任意のビットをクリアする命令　　ANDはクリア
OR命令は任意のビットをセットする命令　　ORはセット
XOR命令は任意のビットを反転させる命令　　XORは反転

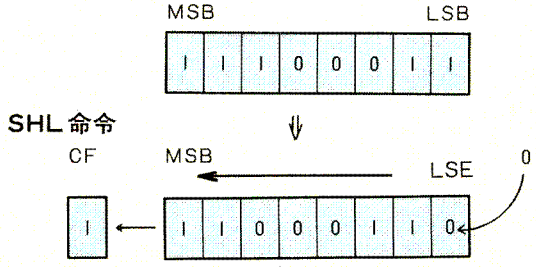
であることがわかります．実行結果は各自で確かめて下さい．

2 シフト・ローテート命令

シフトは桁をずらす命令

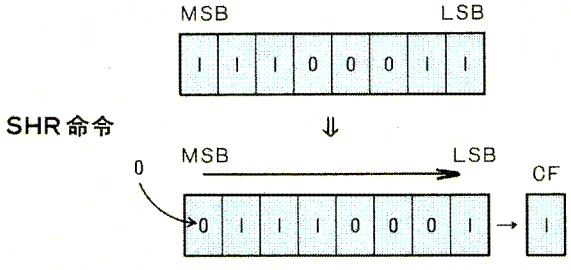
今までのところでは，ひとつひとつのビットの操作を行なってきたわけですが，ここからは，8ビットあるいは16ビットをまとめて動かす命令について説明します．

シフト命令は8ビットあるいは16ビットの各ビットを1桁ずつずらす命令です．具体的には次のようになります．（今後しばらく，8ビットで説明しますが16ビットでも動作は同じです．）



上図のように，SHL命令は，もとのデータの各ビットを左に1ビットずつずらす命令です．最上位ビット（MSB）からはみ出した桁は，CF（キャリー・フラグ）に転送されます．また，空白になった最下位ビット（LSB）には，0が転送されます．

SHR命令は，右向きにずらす命令であり，

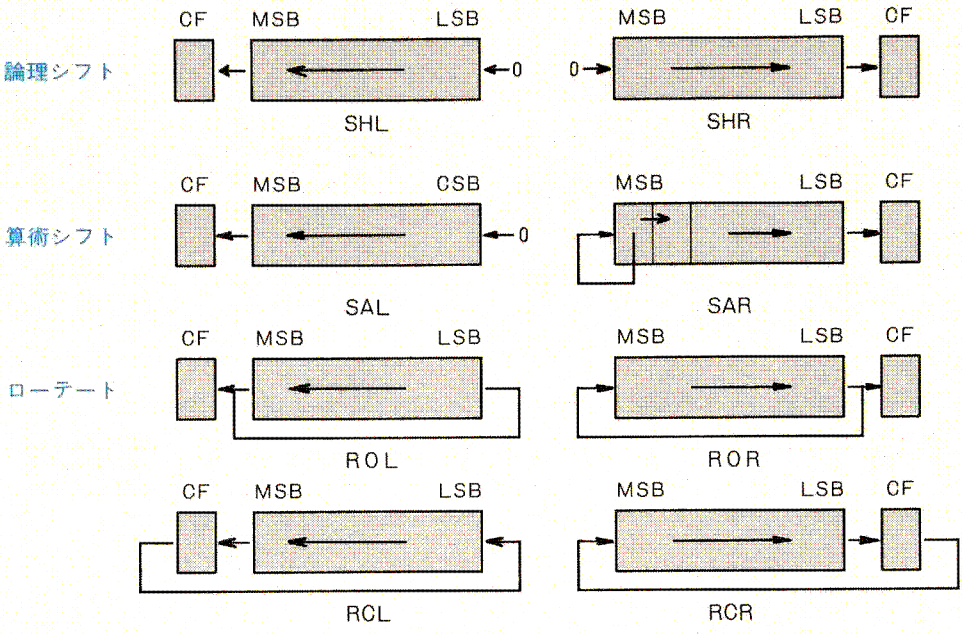


のようになります．

命令の3文字目，Rは右，Lは左

このようにシフト命令は，ビットをずらし，あふれたビットを，キャリー・フラグに転送します．命令の3文字目がLのものは，左（Left）向き，Rのものは右（Right）向きに動きます．

シフト命令・ローテート命令には，いくつかの種類がありますので，まとめて表にしておきます．



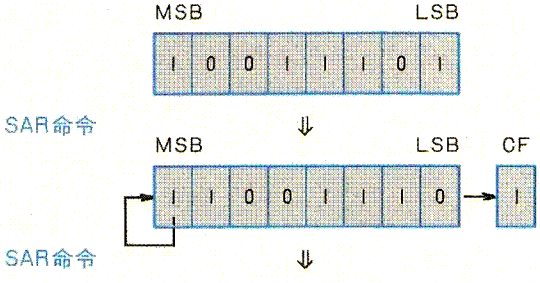
ご覧のとおり，シフト命令は，ただずれていくだけであるのに対し，ローテート命令は，循環しています．ROL 命令は，キャリーフラグを含まずに，最上位ビットのデータは，すぐに最下位ビットに転送されます．その際，キャリー・フラグにも同じデータがセットされます．ROR命令も同様です．当然ですが，8ビットレジスタを用いた場合，この命令を8回くり返せば，もとどおりになります．

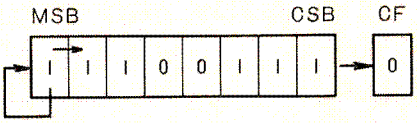
ローテート命令は回転

RCL命令は，キャリーフラグを含んでいっしょに回転させる命令です．したがって最上位ビットがキャリーフラグに，キャリーフラグが最下位ビットに転送されます．RCR命令も同様です．これらの命令は9回繰り返すことによって，もとにもどります．

キャリーを含む命令はまん中がC

少し特殊なのが算術シフトのSAR命令です．この命令は最下位ビットがキャリーフラグに転送されるのは同じですが，最上位ビットには0は入らず，最上位ビットの値が2番めのビットにコピーされ，最上位ビットのデータはそのまま残ります．具体的には次のようになります．





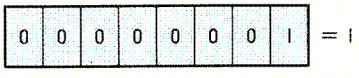
このように1回実行するごとに，最上位ビットのコピーが増えていきます．詳しくは後で説明しますが，これは符号付き数値とみた場合の符合ビットを保存するための命令です．

なお，SAL命令はSHL命令とまったく同じ動作ですから，どちらを使用しても結果は同じです．

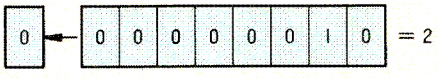
3 乗除算とシフト・ローテート命令

さて，こんなややこしい命令がいったい何のためにあるのでしょうか？実はこの命令は乗除算のためにあるのです．

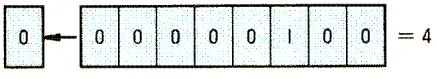
たとえば，8ビットの数値“1”をとって考えてみます．



これを左へ1ビット，シフトします．



もう1回シフトします．

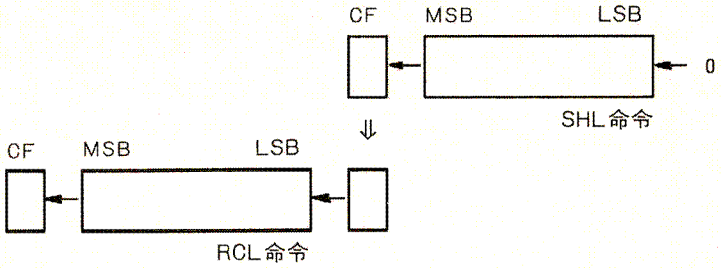


このように，左へ1ビットシフトするごとに，値は2倍，4倍となっていきます．（2進数は，2の累乗を使っているので当然です！）このことを利用すれば，どんなデータも左へ1回シフトすることによって，2倍になります．また，右へシフトすることにより$\frac{1}{2}$にすることもできます．

シフト
ローテート
は乗除算に使う

したがって簡単な乗除算であれば，乗除算命令を用いなくても，シフト・ローテート命令を用いて行なうことができます．前にも述べましたが，乗除算命令の実行には非常に時間がかかるのですが，シフト・ローテート命令は，きわめて高速に行なうことができます．そのため，簡単な乗除算には，よく，シフト・ローテート命令が使用されます．

ところで，なぜシフト命令などではキャリーフラグが使用されるのでしょうか？　賢明な皆さんならおわかりですね．そう，桁上がりを記憶して，多桁の計算を行なうためなのです．たとえば，32ビットの数値を2倍する場合を考えてみます．



このように，SHL命令と，RCL命令を組み合わせることによって，多桁の乗算が行なえます．RCL命令による最上位ビットからの桁上がりは，キャリーフラグに記憶されています．この命令によって，どんなに多桁の数の乗算でよ行なうことができます．したがって，乗算命令では扱うことの難しい多桁の乗算も，容易に行なうことができます．

実際の使用法は，たとえばALレジスタのデータを1ビット左シフトする場合には，

```
SHL     AL, 1
```

のように使用します．8086では一度に何ビットもシフトすることができます．その場合には，あらかじめCLレジスタにシフトするビットを代入きておき，

```
SHL     AL, CL
```

というように使います．

シフト・ローテート命令の整理

SHL (SAL)	reg, 1		レジスタまたはメモリのデータを左へ1ビットシフトする．最上位ビットはキャリーフラグに転送され，最下位ビットには0が補われる．
	mem, 1	*	
	reg, CL		レジスタまたはメモリのデータをCLレジスタの示すビット数だけ左へシフトする．最後に転送されたビットがキャリーフラグに残り，下位ビットから0が補われる．
	mem, CL	*	
SHR	reg, 1		レジスタまたはメモリのデータを右へ1ビットシフトする．最下位ビットがキャリーフラグに転送され，最上位ビットには0が補われる．
	mem, 1	*	

SHL	reg, CL mem, CL	*	レジスタまたはメモリのデータをCLレジスタの示すビット数だけ右へシフトする．最後に転送されたビットがキャリーフラグに残り，上位ビットからは0が補われる．
SAR	reg, CL mem, CL	*	レジスタまたはメモリのデータを右へ1ビットシフトする．最下位ビットがキャリーフラグに転送され，最上位ビットの値は保存されると同時に2番目のビットにコピーされる．
	reg, CL mem, CL	*	レジスタまたはメモリのデータを，CLレジスタの示すビット数だけ右へシフトする．最後に転送されたビットがキャリーフラグに残り，上位ビットからは最上位ビットの値がコピーされていく．
ROL	reg, 1 mem, 1	*	レジスタまたはメモリのデータを左へ1ビット回転する．最上位ビットのデータは，キャリーフラグに転送されると同時に最下位ビットに転送される．
	reg, CL mem, CL	*	レジスタまたはメモリのデータをCLレジスタの示すビット数だけ左へ回転する．最後に転送されたデータがキャリーフラグに残る．
ROR	reg, 1 mem, 1	*	レジスタまたはメモリのデータを右へ1ビット回転する．最下位ビットのデータはキャリーフラグに転送されると同時に最上位ビットに転送される．
	reg, CL mem, CL	*	レジスタまたはメモリとデータをCLレジスタの示すビット数だけ右へ回転する．最後に転送されたデータがキャリーフラグに残る．
RCL	reg, 1 mem, 1	*	レジスタまたはメモリのデータを，キャリーフラグを含めて左へ1ビット回転する．最下位ビットがキャリーフラグに，キャリーフラグが最下位ビットに転送される．
	reg, CL mem, CL	*	レジスタまたはメモリのデータをキャリーフラグを含めて，CLレジスタの示すビット数だけ左へ回転する．最後に転送されたデータがキャリーフラグに残る．
RCR	reg, 1 mem, 1	*	レジスタまたはメモリのデータを，キャリーフラグを含めて右へ1ビット回転する．最下位ビットがキャリーフラグに，キャリーフラグが最上位ビットに転送される．
	reg, CL mem, CL	*	レジスタまたはメモリのデータを，キャリーフラグを含めてCLレジスタの示すビット数だけ右へ回転する．最後に転送されたデータがキャリーフラグに残る．

※これらの命令群では，フラグは複雑に変化する．

例　題

論理演算命令を用いて，AXレジスタの値を10倍するプログラムを作りなさい．AXレジスタにはあらかじめ値として10000（＝2710H）を代入しておきます．

考え方

10d＝1010Bですから，2×2×2＝8倍したものに，2倍したものを加えればよいわけです．桁があふれるので，上位の桁としてBXレジスタを使用します．

解　答

```
0000 B80100          MOV     AX,2710
0003 31DB            XOR     BX,BX
0005 D1E0            SAL     AX,1      ←AXレジスタの値を2倍し，桁上がりをBXレジスタに．
0007 D1D3            RCL     BX,1
0009 89C1            MOV     CX,AX     ←2倍した状態の値を保存しておきます．
000B 89DA            MOV     DX,BX
000D D1E0            SAL     AX,1      ←4倍になります．
000F D1D3            RCL     BX,1
0011 D1E0            SAL     AX,1      ←8倍になります．
0013 D1D3            RCL     BX,1
0015 01C8            ADD     AX,CX     ←8倍と2倍を加えて10倍にします．
0017 11D3            ADC     BX,DX
0019 F4              HLT
```

解　説

結果の上位がBXレジスタに，下位がAXレジスタに格納されます．

とにかく実行してみましょう．

```
hJG0,19
*1B00:0019
hJX
FL ---------
AX 86A0
BX 0001
CX 4E20
```

結果は186A0Hです．これは10進数で100000ですから，正しく求まりました．この方法は，プログラム作成には面倒ですが，実行時間は，乗算命令よりもかなり短くなっています．

2 I/O命令

I／O命令とは，Input／Output命令の略であり，その次のとおり，入出力を行なうための命令です．第1章のはじめのところで，マイクロコンピュータの基本構成について説明しましたが，マイクロコンピュータは大きく分けて，CPU，メモリ，入出力装置，の3つの部分から成っています．CPUにデータを与えて計算をさせたり結果を読み出したりするためには，キーボードやCRTディスプレイなどの出力装置との間でデータを交換することが必要です．そのとき用いるのがI/O命令です．

Input/Output命令

ところで，CPUとメモリとの間でのデータの転送はアドレスを指定して行ないましたが，CPUと入出力装置との間でのデータの交換はどのように行なわれるのでしょうか．

16ページの図にあるように，PC－9801には様々な入出力装置があります．実はこれらの入出力装置には，I／Oポート番号が付けられていて，CPUはその番号を指定して入出力を行なうのです．そのための命令がI／O命令です．CPUと同じ基板の上に乗っているとはいえ，キーボード・インターフェスや，グラフィック・ディスプレイ・コントローラなどはCPUから見れば外部の装置であり，それぞれが独立した“頭脳”を持ったLSIによってコントロールされています．したがってそれぞれを使用するためには，まずCPUが入出力装置にコマンドを送り，その後にデータの交換を行なう，といった手続をふむ必要があります．これらのコマンドの使用法は各入出力装置によって違っていますから，ここではその使用法は掲げません．

I／Oポート番号

LSIは頭脳を持つ

1 I/O命令の使用法

それではI/O命令の使用法について説明しましょう．I/O命令自体の使用法はきわめて簡単です．

ポート番号と送るデータを指定する

まず，OUT命令ですが，OUT命令は入出力装置にデータやコマンドを転送するための命令です．送るべきデータが8ビットならばALレジスタに，16ビットならばAXレジスタにデータをセットした上で，

```
OUT　　37, AL
```

というように使用します．これで，37H番のポートに接続されている入出力装置にALレジスタのデータが転送されます．

次にIN命令ですが，IN命令ではデータを格納すべきレジスタとしてAL又はAXレジスタを選び，ポート番号を指定します．

```
IN　　AL, 41
```

これで，41H番のポートのデータがALレジスタに格納されました．

なお，I/Oポートの番号としては0H～0FFHまでが選べるほかに，DXレジスタにポート番号をセットして，

```
OUT　　DX, AL
```

というように使うことで,0H～0FFFFHまでのポート番号を指定することもできますが，PC-9801では，そんなに多くのI/Oポートを備えていませんから，この命令を使うことはあまりないでしょう．

実際にI/Oポートを使用するためには，ハードウェアの知識が少なからず要求されます．現実には皆さんがI/Oポートをコントロールする必要は，ほとんどなく，入出力のために必要なプログラムはすでにBASIC・ROMとして存在していますから，これを使用すれば用がたります．このような，入出力のためのプログラムを総称してBIOS（Basic Input Output System）と呼んでいます．

BIOS(Basic Input Output System)

ここでは簡単な例題を掲げるに留めましょう．

例題

I/O命令を用いて，ブザーを鳴らす命令を作りなさい．

考え方

ブザーをコントロールするためには，ポート37Hにコマンドを送ります．06Hがブザーを鳴らすコマンド，07Hがブザーを止めるコマンドです．

解　答

```
0000 B006          MOV     AL,06      ←ブザーを鳴らすコマンド.
0002 E637          OUT     37,AL
0004 B308          MOV     BL,08
0006 B90000        MOV     CX,0000
0009 E2FE          LOOP    0009       ←約2秒間待つプログラム.
000B FECB          DEC     BL
000D 75F7          JNE     0006
000F B007          MOV     AL,07
0011 BA3700        MOV     DX,0037    ←ブザーを止めるコマンド. こうした使用法も可能.
0014 EE            OUT     DX,AL
0015 F4            HLT
```

解　説

結果は自分で確かめて下さい．ウエイトルーチンを入れないと，音が短くて聞こえなくなってしまいます．

ウエイトルーチン

IN.OUT命令の整理

IN	acc, ポート番号	……指定したポートからデータをALまたはAXレジスタに読込む.
	acc, DX	……DXレジスタで指定したポートからデータをALまたはAXレジスタに読み込む.
OUT	ポート番号, acc	……指定したポートにALまたはAXレジスタのデーを送り出す.
	DX, acc	……DXレジスタで指定したポートにALレジスタのデータを送り出す.

※accは，アキュムレータ・レジスタすなわちALまたはAXレジスタのこと.
※ポート番号は0H～0FFH，DXレジスタの値は0H～0FFFFH.
※この命令群ではフラグは変化しない.

なお，80系のCPUは入出力用にこのようなI/O命令を持っていますが，メモリ空間上に入出力装置を割り当てることも可能です．この場合はそのメモリのアドレスをリード，ライトすることにより，入出力装置を制御することができます．68系のCPUはその方法を用いています．I/O命令のデータも当然バスを使用して交換されるわけですから，80系のCPUでは現在のデータがメモリに対するものなのか，入出力装置に対するものなのかを，別の信号線(M/$\overline{IO}$)でCPUから周辺装置に知らせています．

I/O・マップド・I/O
メモリ・マップド・メモリ

3 その他の命令

さて，いよいよマシン語の勉強も終わりに近づきました．最後に，今までの説明ではふれられなかった命令をまとめて，マシン語学習のしめくくりとしたいと思います．

1 2進化10進数

コンピュータが扱うことのできる数は２進数であり，２進数を人間がわかりやすく表現するために16進数という数を導入したことは前に説明したとおりです．しかし，やはり16進数はわかりにくいものです．そこで，2進数の形を用いて10進数を表すことはできないのか？ということで生み出されたのが２進化10進数という表現法です．

この方法は，２進数の４ビットで10進数の１桁を表す方法です．４ビットあれば０からＦまでの数値を表わすことができるのですが，そのうちの０から９までを使用し，ＡからＦまでの数値は使用しないことに決めておきます．そうすれば２進数の４ビットが10進数の１桁になりますから，コンピュータで10進数を扱うことができます．

たとえば，10進数の17は16進数では，

```
0001 0001  =  11
```

ですが，２進化10進数では，

```
0001 0111  =  17
```

となります．

このような方法を用いれば，10進数を2進数の形式を借りて表現ができます．なお，２進化10進数のことを**BCD コード**(Binary Coded Decimal)といいます．

BCD コード

2 10進補正命令

ところでコンピュータは，何もこの2進化10進数のための特別な演算命令を備えているわけではありません．そこで，2進化10進数の計算の際にちょっと困ったことが起こります．

例をあげてみましょう．たとえば，10進数の17＋13という計算を2進化10進数を用いて表わし，通常のADD命令を実行すると，

```
      0 0 0 1 | 0 1 1 1      1 7
+     0 0 0 1 | 0 0 1 1      1 3
-------------------------------------
      0 0 1 0 | 1 0 1 0      2 A   ?
```

ということになります．2進化10進数にはAという数はありません．どうしましょうか？

ここで用いられるのが10進補正命令です．この命令を用いると，A以上の結果が出た場合には自動的に桁をくり上げて，正しい2進化10進数に変換されす．

10進補正命令DAA(Decimal Adjust for Additor)で前の結果を補正すると，

```
      0 0 1 0 | 1 0 1 0      2 A
+     0 0 0 0 | 0 1 1 0      0 6   ←10進補正命令により自動的に加えられる．
-------------------------------------
      0 0 1 1 | 0 0 0 0      3 0
```

となり，正しい結果の30を得ることができます．10進補正命令は，2進化10進数で表わせない部分だけに自動的に6を加えることが，正しい2進化10進数の結果を得るという命令です．(なぜ6かというと，16進数と10進数は1桁が6だけ違うからです) 10進補正命令は，演算命令の直後に置かれます．なお，**10進補正命令は，ＡＬレジスタのみに使用されます．**

10進補正命令はＡＬレジスタを使用

上記の例をプログラムにしてみましょう．

```
MOV AL,17
ADD AL,13
DAA
```

DAA命令は，4ビット単位の2進化10進数の加算結果を補正する命令です．なぜ4ビット単位と言ったかというと，実は2進化10進数には8ビットで10進

数の１桁を表す方法もあるからです．つまり，8ビットの上位４ビットは常に０として，１バイトが10進数の１桁を表わすようにしたものです．もったいない使い方ですが，入出力の１バイトが１桁に対応するため，わかりやすい方法でもあります．

このように，８ビットが10進数の１桁を表わすような２進化10進数を，**アンパックド10進数**といい，４ビットが１桁すなわち８ビットが10進数の２桁を表わすような２進化10進数を**パックド10進数**と呼び，それぞれに違った補正命令が用意されています．

アンパックド10進数

パックド10進数

これに対応する命令を表にしておきましょう．

	加算後補正	減算後補正
アンパックド	AAA (ASCII Adjust for Addition)	AAS (ASCII Adjust for Subtraction)
パックド	DAA (Decimal Adjust for Addition)	DAS (Decimal Adjust for Subtraction)

10進数補正命令には，このほかにアンパックド10進数どうしの乗算後の補正を行なうAAM（ASCII Adjust for Multiply）命令，および，アンパックド10進数どうしの除算の前に被除数の補正を行なうAAD（ASCII Adjust for Division）命令があります．

AAM命令

ADD命令

なお，これらの命令では，桁上がりや借りを正じた場合にはキャリーフラグがセットされます．多桁の計算を行なう場合には，１バイト計算するごとに補正を行なわなければなりません．

10進補正命令の整理

AAA ……AL レジスタのアンパックド10進数に対し加算後補正を行なう
DAA ……AL レジスタのパックド10進数に対し加算後補正を行なう
AAS ……AL レジスタのアンパックド10進数に対し減算後補正を行なう
DAS ……AL レジスタのパックド10進数に対し減算後補正を行なう
AAM ……AL レジスタのアンパックド10進数に対し乗算後補正を行なう
AAD ……AL レジスタのアンパックド10進数（被除数）に対し，除算前補正を行なう．

※これらの命令では結果によってフラグが変化する．

例　題

10進数値1234ｄと5678ｄとを２進化10進数を用いて加算しなさい．

```
0000 BB3412          MOV     BX,1234
0003 B97856          MOV     CX,5678
0006 88D8            MOV     AL,BL
0008 00C8            ADD     AL,CL
000A 27              DAA
000B 88C3            MOV     BL,AL
000D 88F8            MOV     AL,BH
000F 10E8            ADC     AL,CH
0011 27              DAA
0012 88C7            MOV     BH,AL
0014 F4              HLT
```

ALレジスタを用いて，下位バイトを加算後，10進補正を行います．

ALレジスタを用いて上位バイトを加算後，10進補正を正を行います．前の結果のキャリーも加えます．

解　説

ＢＸレジスタに結果を納めます．補正命令はＡＬレジスタに対してしか行なえませんから，すべての演算はＡＬレジスタを用いて行なわなければなりません．補正後に残ったキャリーを，上位の桁に加えることを忘れないで下さい．（MOV 命令でフラグは変化しない）

実行結果は，

```
hJG0,14
*1B00:0014
hJX
FL -------P-
AX 0469
BX 6912
CX 5678
```

となり，正しい加算結果6912がＢＸレジスタに格納されています．

3 特殊な転送命令

以下の命令は，やや特殊な転送命令ですが，テーブルを参照してデータをさがす場合などに使用されます．

LEA (Load effective address) 命令

この命令は，２番めのオペランドの示す，**アドレスの値**を第１オペランドに転送する命令です．つまり，アドレスそのものの値を求める命令であり，そのアドレスに格納されている値は関係がありません．

たとえば，

```
MOV     SI, 1234
LEA     BX, [SI]
```

とすると，BXレジスタには，SIの値である1234Hが転送されます．

```
MOV     BX, 1000
MOV     SI, 2000
LEA     DI, 3000[BX+SI]
```

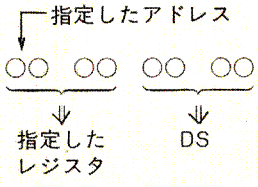
とすると，DIレジスタには，1000H＋2000H＋3000H＝6000Hが転送されます．

第2オペランドは，メモリポインタ間接アドレス指定でなければなりません．上に示したように，オペランドにおけるアドレッシングによって指定されるメモリの**アドレス値**を**実効アドレス**（EA：Effective Address）といいます．

実効アドレス(EA)

LDS(Load Data Segment register)命令

この命令は，指定したアドレスからの4バイトを，次のように転送します．

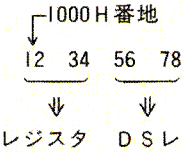


第1オペランドが，データを転送されるレジスタを，第2オペランドがデータのあるアドレスを示します．そこでたとえば，

```
LDS     SI , [1000]
```

という命令を実行したとします．1000H番地からに次のようなデータが入っていたとすれば，

ストリング命令
LODS
MOVS等の前
に使うとよい



SIレジスタの値が3412Hに，DSレジスタの値が7856Hになります．

この命令は，あらかじめ用意されたアドレステーブル（この場合は1000H番地）を参照して，データを別のセグメントから取り出す際に使用されます．

LES（Load Extra Segment register）命令も，DSがESにかわっただけで，動作は同じです．

LES命令

XLAT(TransLATe)命令

この命令は，ＢＸレジスタとＡＬレジスタを用いたアドレス指定により，データを取り出す命令です．この命令を実行する前にはあらかじめ，ＢＸレジスタとＡＬレジスタに値をセットしておかなければなりません．この命令を実行すると，ＢＸレジスタの示すアドレスから，ＡＬレジスタの示すバイト分だけ後にあるバイトのデータを読み込み，ＡＬレジスタに格納します．

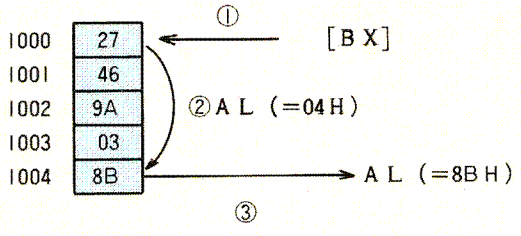
たとえば，1000Ｈ番地以後に次のようなデータが格納されているとします．

アドレス	メモリ
1000	27
1001	46
1002	9A
1003	03
1004	8B

この際，次の命令を実行します．

```
MOV     BX, 1000
MOV     AL, 04
XLAT
```

すると，次の図のような順にデータが捜し出され，ＡＬレジスタに83Ｈというデータが格納されて終了します．



この命令は，あらかじめデータのテーブルを作っておき，必要に応じて，指定されたNo.のデータを持ってくるような操作に使用されます．

特殊な転送命令の整理

LEA	reg, mem	……memのアドレッシングによるアドレスの値をレジスタに転送します．
LDS	reg, mem	……メモリ上の４バイトデータの下位２バイトを指定したレジスタに，上位２バイトをＤＳに転送します．
LES	reg, mem	……メモリ上の４バイトデータの下位２バイトを指定したレジスタに，上位２バイトをＥＳに転送します．
XLAT		……ＢＸレジスタの示すアドレスに，ＡＬレジスタの値を加えたアドレスの内容を読み出し，ＡＬレジスタに格納します．

※これらの命令群ではフラグは変化しません．

4 CPU制御命令

最後の命令は，文字通りCPUを制御する命令です．これらの命令の中には，CPUが，他のLSIと同期をとって命令を実行する場合などに使用されるものが含まれています．

ESC(ESCape) 命令

この命令は8086が8087数値演算プロセッサを制御するための命令です．8086はこの命令を読み込むと，命令の番号および命令中に指定されたアドレスから読み出したデータを，8087に転送します．8086自身はアドレスを計算して，データをデータバス上に乗せるだけで，何も変化しません．この命令により，8087に命令やデータを送ることができます．

WAIT(WAIT) 命令

この命令を実行すると，8087が演算を終了するまで実行を停止して待ちます．正確に言えば，CPUのTEST端子に信号が与えられるまで実行を停止します．このTEST端子は8087に接続されています．WAIT命令実行中でも，外部割り込みを受け付けることができます（IFがセットされている場合）が，終了後再びWAIT状態になります．この命令は，CPUの動作を8087など他のLSIに同期させるために使用します．

LOCK(LOCK) 命令

この命令を実行すると，これに続く1バイトの命令の実行中に他の周辺装置がバスを使用することを禁止します．8086をマキシムモードで使用するときに，いくつかの周辺装置がバスを使用していますが，この命令の次の命令の実行中は，他の周辺装置からのバス使用要求は受け付けられなくなります．

この命令は，CPUがメモリを連続して操作中であることを示すためのメモリ上のアドレス（セマフォという）をセット・リセットする際に使用します．XCHG命令の実行中に，同じアドレス（セマフォ）を他のLSIが操作しないために必要となります．セマフォは，連続して操作しなければならないメモリ空間を操作中に，他LSIが同じメモリを書き換えないようにするための“操作中”を示すフラグの役割をはたすメモリのことです．

セマフォ

HLT(HaLT) 命令

この命令を実行するとCPUは実行を停止します．イネーブル（許可）された外部割り込みが発生すると，割り込み処理終了後，次の命令から実行を再開します．もちろんリセットによって抜け出すこともできます．この命令は，割り込みを待って実行を再開する場合などに使用します．

NOP(NoOPeration) 命令

この命令は，何もせずに次の命令に進む命令です．何もしない命令が何のためにあるのかというと，プログラムのバイト数を調整したり，実行速度の遅い入出力装置などとタイミングをあわせるための“待ち時間”を捜入したりするために用いられます．

CPU制御命令の整理

ESC	n，reg n，mem	……8087にn番めのコマンドを送るとともに，レジスタまたはメモリの内容を8087に転送します．
WAIT		……8087の演算が終わるまで実行を停止して待ちます．
LOCK		……この命令に続く1命令の実行中，他の周辺機器のバスの使用を禁止します．
HLT		……イネーブルされた外部割込みがあるまで実行を停止して待ちます．
NOP		……何も実行せずに次の命令に進みます．

※これらの命令群でフラグな変化しない．

巻末

APPENDIX

命令一覧表

ハンド・アセンブルと
ハンド・ディスアセンブル

命令一覧表

命令一覧表の読み方

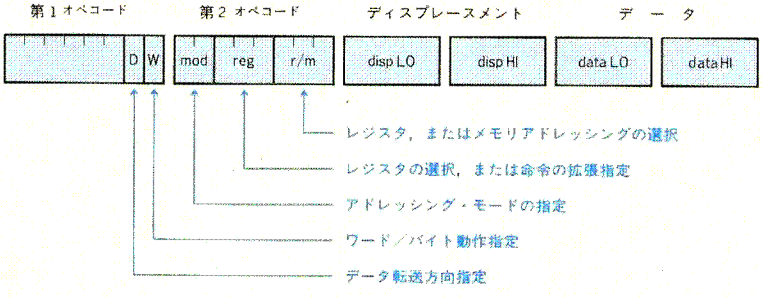
命令はABC順に並べてあります．必要な命令とその機能を調べるためには，インストラクションセットを見て下さい．使用されている用語は，**表2**および**表3**に掲げてあります．命令の基本構成は**表1**のようになっています．命令によってバイト数が変化します．8086の命令の多くは，第1オペランド中に，バイト操作か，ワード操作かが示されています．第2オペコードを持つか否か，データを持つか否かは第1オペコードによって決定されます．

第2オペコードを持つものの場合，操作の行なわれるレジスタおよびメモリがここで指定されます．ディスプレースメントを持つか否かは，第2オペコードによって決定されます．操作の行なわれるメモリまたはレジスタは，modとr／mの組み合わせによって指定されます(**表4**参照)．レジスタ・メモリ，またはレジスタ・レジスタ間で操作が行なわれる場合には，regによってもう一方のレジスタが指定されます(**表3**参照)．また，一つのメモリまたはレジスタのみに対して操作が行なわれる命令の場合には，regが命令の種類を指定することがあります(オペランド一覧表参照)．これらによって指定されるレジスタおよびメモリのビット数は，第1オペコードのWビットによって決定されます．

バイト数は命令の長さを，クロック数は，命令の実行に必要なクロック数を，表わしています．これらは，データやディスプレースメントの有無で長さが変化します．

フラグはオペレーションの説明の中で，レジスタについて直接演算を行なうものに対しては(AX)+(BX)のように(　　)を付けて表わしました(3章2節1の注参照)．したがって本文中のAX←AX+BXのような表記は厳密な表記ではないので，すべて(AX)←(AX)+(BX)などに置き換えます．(AX)などは(reg)に，従来の1234[BX]などは(mem)にそれぞれ表現を変えます．

1　命令の基本構成



2　オペ・コードで使用される省略語の表

W	ワード／バイト・フィールド	（1ビット）
reg	レジスタ・フィールド	（3ビット）
r/m	レジスタ・メモリ・フィールド	（3ビット）
mod	モード・フィールド	（2ビット）
sreg	セグメント・レジスタ・フィールド	（3ビット）

3　選択されるレジスタの表（reg フィールドで指定される）

reg　または r/m（mod = 11 のとき）	w = 0 （バイト動作）	w = 1 （ワード動作）	セグメント （sreg）
000	AL	AX	ES
001	CL	CX	CS
010	DL	DX	SS
011	BL	BX	DS
100	AH	SP	—
101	CH	BP	—
110	DH	SI	—
111	BH	DI	—

4　メモリ・アドレッシングの表（mod，r／m フィールドで指定される）

r/m ＼ mod	00	01	10	11
000	[BX＋SI]	disp 8 [BX＋SI]	disp 16 [BX＋SI]	上表参照
001	[BX＋DI]	disp 8 [BX＋DI]	disp 16 [BX＋DI]	
010	[BP＋SI]	disp 8 [BP＋SI]	disp 16 [BX＋SI]	
011	[BP＋DI]	disp 8 [BP＋DI]	disp 16 [BP＋DI]	
100	[SI]	disp 8 [SI]	disp 16 [SI]	
101	[DI]	disp 8 [DI]	disp 16 [DI]	
110	DIRECT	disp 8 [BP]	disp 16 [BP]	
111	[BX]	disp 8 [BX]	disp 16 [BX]	

5　オペランドおよびオペレーション解説中で使用される省略語の表

reg	8 or 16ビット汎用レジスタ
reg 8	8 ビット汎用レジスタ
reg 16	16ビット汎用レジスタ
mem	8 or 16ビットメモリ・ロケーション （注1）
mem 8	8 ビットメモリ・ロケーション
mem 16	16ビットメモリ・ロケーション
mem 32	32ビットメモリ・ロケーション
imm	0 ～ 0 FFFFH までの範囲の数値
imm 8	0 ～ 0 FFH までの範囲の数値
data	8 or 16 ビットデータ
data 8	8 ビットデータ
data 16	16ビットデータ
[data16←data8]	16ビットに符号拡張されたデータ
acc	AX レジスタ又は AL レジスタ
source-table	256バイトの変換テーブル
src-string	SI レジスタによってアドレスされるストリングの名前
dst-string	DI レジスタによってアドレスされるストリングの名前
short-label	命令の終わりから －128 d～＋127 d バイトの範囲のラベル（アドレス）
near-label	現在のコード・セグメント内のラベル（アドレス）
far-label	別のコード・セグメント内のラベル（アドレス）
near-proc	現在のコード・セグメント内のプロシージャ（サブルーチン名）
far-proc	別のコード・セグメント内のプロシージャ（サブルーチン名）
memptr 16	現在のコード・セグメント内にジャンプまたはコールする先のオフセット・アドレスの書かれているメモリのロケーション.
memptr 32	別のコード・セグメント内にジャンプまたはコールする先のオフセット・アドレスとセグメントベース・アドレスの書かれているメモリのロケーション.
regptr 16	現在のコード・セグメント内にジャンプまたはコールする先のオフセット・アドレスの書かれている 16 ビット汎用レジスタ.
repeat	ストリング命令のリピートプリフィックス
optional-pop-value	RET 命令において，スタックから捨てられるバイト数（0～64 K，普通は偶数）
(　)	(　) 内のレジスタまたはメモリの内容
((　))	((　)) 内のメモリ（間接指定された）の内容（3章2節参照）
:	連続したバイトのアドレス（3章2節参照）
disp	8 or 16ビットディスプレースメント（5章1節10項参照）
temp	8 ビットテンポラリ（一時）レジスタ
tmpcf	テンポラリキャリー・フラグ
+, −, ×, /, %	加減乗除，および剰余
&, V, ¥	論理積，論理和，排他的論理和

（注1）様々なアドレッシング・モードによる，メモリのアドレスを指定するための記述.
（注2）この表の表記は ASM86 を考慮している. 一覧表末の注参照

6　フラグの動作

ブ　ラ　ン　ク	変化しない.
0	0にクリア（リセット）される.
1	1にセットされる.
X	結果に従ってセットまたはクリアされる.
U	不定になる.
R	以前に退避した値がリストアされる.

7　命令のクロック数

クロック数の項で「+EA」と書かれているものは，アドレッシングモードによって異なるものです．下の表に従ってEAの位を求めて，クロック数に加算して下さい．

ダイレクト・アドレス	6
[BX]，[BP]，[SI]，[DI]	5
disp [BX]，disp [BP]，disp [SI]，disp [DI]	9
[BX+SI]，[BP+DI]	7
[BX+DI]，[BP+SI]	8
disp [BX+SI]，disp [BP+DI]	11
disp [BX+DI]，disp [BP+SI]	12

※ セグメント・オーバーライドに対しては2クロックを加える．
※ 8088でワードデータを扱う際には，1回ワードデータを扱う毎に4クロックを加える．

命令一覧表（インストラクションセット）

ニーモニック	オペランド	オペレーション・コード 76543210	オペレーション・コード 76 543 210	第1オペコード バイト/ワード	バイト数	クロック数	オペレーション	フラグ O	S	Z	A	P	C
AAA		00110111	—	37	1	4	if((AL)&0FH)>9or(AF)=1then(AL)←(AL)+6,(AH)←(AH)+1,(AF)←1,(CF)←(AF),(AL)←(AL)&0FH	U	U	U	×	U	×
AAD		11010101	00 001 010	D5,0A	2	60	(AL)←(AH)*0AH+(AL) (AH)←0	U	×	×	U	×	U
AAM		11010100	00 001 010	D5,0A	2	83	(AH)←(AL)／0AH (AL)←(AL)%0AH	U	×	×	U	×	U
AAS		00111111	—	3F	1	4	if((AL)&0FH)>9 or (AF)=1 then (AL)←(AL)−6,(AH)←(AH)−1,(AF)←1 (CF)←(AF),(AL)←(AL)&0FH	U	U	U	×	U	×
ADC	reg,reg	0001000W	11 reg r/m	10/11	2	3	(reg)←(reg)+(reg)+(CF)	×	×	×	×	×	×
	mem,reg	0001000W	mod reg r/m	10/11	2−4	16+EA	(mem)←(mem)+(reg)+(CF)	×	×	×	×	×	×
	reg,mem	0001001W	mod reg r/m	12/13	2−4	9+EA	(reg)←(reg)+(mem)+(CF)	×	×	×	×	×	×
	reg,imm	1000000W	11 010 r/m	80/81	3−4	4	(reg)←(reg)+data+(CF)	×	×	×	×	×	×
		1000001W	〃	82/83	3	4	ifw=0 (reg)←(reg)+data8+(CF) ifw=1 (reg)←(reg)+[data16←data8]+(CF)	×	×	×	×	×	×
	mem,imm	1000000W	mod 010 r/m	80/81	3−6	17+EA	(mem)←(mem)+data+(CF)	×	×	×	×	×	×
		1000001W	〃	82/83	3−5	17+EA	ifw=0 (mem)←(mem)+data+(CF) ifw=1 (mem)←(mem)+[data16←data8]+(CF)	×	×	×	×	×	×
	acc,imm	0001010W	—	14/15	2−3	4	ifW=0 (AL)←(AL)+data+(CF) ifW=1 (AX)←(AX)+data+(CF)	×	×	×	×	×	×
ADD	reg,reg	0000000W	11 reg r/m	00/01	2	3	(reg)←(reg)+(reg)	×	×	×	×	×	×
	mem,reg	0000000W	mod reg r/m	00/01	2−4	16+EA	(mem)←(mem)+(reg)	×	×	×	×	×	×
	reg,mem	0000001W	mod reg r/m	02/03	2−4	9+EA	(reg)←(reg)+(mem)	×	×	×	×	×	×
	reg,imm	1000000W	11 000 r/m	80/81	3−4	4	(reg)←(reg)+data	×	×	×	×	×	×
		1000001W	11 000 r/m	82/83	3	4	ifW=0 (reg)←(reg)+data8 ifW=1 (reg)←(reg)+[data16←data8]	×	×	×	×	×	×
	mem,imm	1000000W	mod 000 r/m	80/81	3−6	17+EA	(mem)←(mem)+data	×	×	×	×	×	×
		1000001W	mod 000 r/m	82/83	3−5	17+EA	ifW=0 (mem)←(mem)+data ifW=1 (mem)←(mem)+[data16←data8]	×	×	×	×	×	×
	acc,imm	0000010W	—	08/09	2−3	4	ifW=0 (AL)←(AL)+data 8 ifW=1 (AX)←(AX)+data 16	×	×	×	×	×	×
AND	reg,reg	0010000W	11 reg r/m	20/21	2	3	(reg)←(reg)&(reg)	○	×	×	U	×	○
	mem,reg	0010000W	mod reg r/m	20/21	2−4	16+EA	(mem)←(mem)&(reg)	○	×	×	U	×	○
	reg,mem	0010001W	mod reg r/m	22/23	2−4	9+EA	(reg)←(reg)&(mem)	○	×	×	U	×	○
	reg,imm	1000000W	11 100 r/m	80/81	3−4	4	(reg)←(reg)& data	○	×	×	U	×	○
	mem,imm	1000000W	mod 100 r/m	80/81	3−6	17+EA	(mem)←(mem)& data	○	×	×	U	×	○
	acc,imm	0010010W	—	24/25	2−3	4	ifW=0 (AL)←(AL)& data ifW=1 (AX)←(AX)& data	○	×	×	U	×	○
CALL	near-proc	11101000	—	E8	3	19	(SP)←(SP)−2 ((SP)+1:(SP))←(IP),(IP)←(IP)+disp						
	regptr 16	11111111	11 010 r/m	FF	2	16	(SP)←(SP)−2 ((SP)+1:(SP))←(IP),(IP)←(regptr16)						
	memptr 16	11111111	mod 010 r/m	FF	2−4	21+EA	(SP)←(SP)−2 ((SP)+1:(SP))←(IP),(IP)←(memptr16)						
	far-proc	10011010	—	9A	5	28	(SP)←(SP)−2 ((SP)+1:(SP))←(CS),(CS)←seg (SP)←(SP)−2 ((SP)+1:(SP))←(IP),(IP)←offset						
	memptr 32	11111111	mod 011 r/m	FF	2−4	37+EA	(SP)←(SP)−2 ((SP)+1:(SP))←(CS), (CS)←(memptr32) (SP)←(SP)−2 ((SP)+1:(SP))←(IP),(IP)←(memptr32)						
CBW	—	10011000	—	98	1	2	if(AL)<80H then(AH)←0 else (AH)←FFH						
CLC	—	11111000	—	F8	1	2	(CF)←0						○
CLD	—	11111100	—	FC	1	2	(DF)←0						
CLI	—	11111010	—	FA	1	2	(IF)←0						
CMC	—	11110101	—	F5	1	2	(CF)←($\overline{CF}$)						×
CMP	reg,reg	0011100W	11 reg r/m	38/39	2	3	(reg)−(reg)	×	×	×	×	×	×
	mem,reg	0011100W	mod reg r/m	38/39	2−4	9+EA	(mem)−(reg)	×	×	×	×	×	×
	reg,mem	0011101W	mod reg r/m	3A/3B	2−4	9+EA	(reg)−(mem)	×	×	×	×	×	×
	reg,imm	1000000W	11 111 r/m	80/81	3−4	4	(reg)−data	×	×	×	×	×	×
		1000000W	11 111 r/m	82/83	3	4	(reg)−[data16←data8]	×	×	×	×	×	×
	mem,imm	1000000W	mod 111 r/m	80/81	3−6	10+EA	(mem)−data	×	×	×	×	×	×
		1000001W	mod 111 r/m	82/83	3−5	10+EA	(mem)−[data16←data8]	×	×	×	×	×	×
	acc,imm	0011110W	—	3D/3E	2−3	4	ifW=0 (AL)−data8 ifW=1 (AX)−data16	×	×	×	×	×	×
CMPSB CMPSW	(dst-string, src-string)	1010010W	—	A4/A5	1	22or 9+ 22*rep	ifW=0 ((SI))−((DI)) if(DF)=0 then(SI)←(SI)+1,(DI)←(DI)+1 else (SI)←(SI)−1,(DI)←(DI)−1 ifW=1 ((SI)+1:(SI))−((DI)+1:(DI)) if(DF)=0 then(SI)←(SI)+2,(DI)←(DI)+2	×	×	×	×	×	×

ニーモニック	オペランド	オペレーション・コード		第1オペコード バイト/ワード	バイト数	クロック数	オペレーション	フラグ					
		76543210	76 543 210					O	S	Z	A	P	C
							else (SI)←(SI)−2, (DI)←(DI)−2						
CWD	—	10011001	—	AA	1	1	if(AX)<8000H then(DX)←0 else (DX)←FFFFH						
DAA	—	00100111	—	27	1	4	if((AL)&0FH)>9or(AF)=1 then (AL)←(AL)+6, (CF)←(AF)∨(CF), (AF)←1 if(AL)>9FHor(CF)=1 then (AL)←(AL)+60H, (CF)←1	×	×	×	×	×	×
DAS	—	00101111	—	2F	1	4	if((AL)&0FH)>9 or (AF)=1 then (AL)←(AL)−6, (CF)←(AF)∨(CF), (AF)←1 if(AL)>9FH or (CF)=1 then (AL)←(AL)−60H, (CF)←1	U	×	×	×	×	×
DEC	reg8	11111110	11 001 r/m	FE	2	3	(reg8)←(reg8)−1	×	×	×	×	×	
	mem	1111111W	mod 001 r/m	FE/FF	2−4	15+EA	(mem)←(mem)−1	×	×	×	×	×	
	reg16	01001reg	—	4X別表	1	2	(reg16)←(reg16)−1	×	×	×	×	×	
DIV	reg8	11110110	11 110 r/m	F6	2	80−90	(temp)←(AX) if(temp)/(reg8)>FFH then (SP)←(SP)−2, ((SP)+1 : (SP))←FLAGS, (IF)←0, (TF)←0, (SP)←(SP)−2, ((SP)+1 : (SP))←(CS), (CS)←(2) (SP)←(SP)−2, ((SP)+1 : (SP))←(IP), (IP)←(0) else (AL)←(temp)/(reg8), (AH)←(temp)%(reg8)	U	U	U	U	U	U
	mem8	11110110	mod 110 r/m	F6	2−4	(86−96) +EA	(temp)←(AX) if(temp)/(mem8)>FFH then (SP)←(SP)−2, ((SP)+1 : (SP))←FLAGS, (IF)←0, (TF)←0, (SP)←(SP)−2, ((SP)+1 : (SP))←(CS), (CS)←(2) (SP)←(SP)−2, ((SP)+1 : (SP))←(IP), (IP)←(0) else (AL)←(temp)/(mem8), (AH)←(temp)%(mem8)	U	U	U	U	U	U
	reg16	11110111	11 110 r/m	F7	2	144−162	(temp)←(DX : AX) if(temp)/(reg16)>FFFFH then (SP)←(SP)−2, ((SP)+1 : (SP))←FLAGS, (IF)←0, (TF)←0, (SP)←(SP)−2, ((SP)+1 : (SP))←(CS), (CS)←(2) (SP)←(SP)−2, ((SP)+1 : (SP))←(IP), (IP)←(0) else (AL)←(temp)/(mem8) (AH)←(temp)%(mem8)	U	U	U	U	U	U
	mem16	11110111	mod 110 r/m	F7	2−4	(150−168) +EA	(temp)←(DX : AX) if(temp)/(mem16)>FFFFH then (SP)←(SP)−2, ((SP)+1 : (SP))←FLAGS, (IF)←0, (TF)←0, (SP)←(SP)−2, ((SP)+1 : (SP))←(CS), (CS)←(2) (SP)←(SP)−2, ((SP)+1 : (SP))←(IP), (IP)←(0) else (AX)←(temp)/(mem16), (DX)←(temp)%(mem16)	U	U	U	U	U	U
ESC	imm, reg	11011XXX	11 YYY r/m	DX別表	2	2	CPU escape data bus←(reg)						
	imm, mem	11011XXX	mod YYY r/m	DX別表	2−4	8+EA	CPU escape data bus←(mem)						
HLT	—	11110100	—	F4	1	2	CPU halt						
IDIV	reg8	11110110	11 111 r/m	F6	2	101−112	(temp)←(AX) if(temp)/(reg8)>0 and(temp)/(reg8) >7FH or(temp)/(reg8)>0 and (temp)/(reg8)<0−7FH−1 then (SP)←(SP)−2, ((SP)+1 : (SP))←FLAGS, (IF)←0, (TF)←0, (SP)←(SP)−2, ((SP)+1 : (SP))←(CS), (CS)←(2) (SP)←(SP)−2, ((SP)+1 : (SP))←(IP), (IP)←(0) else (AL)←(temp)/(reg8), (AH)←(temp)%(reg8)	U	U	U	U	U	U
	mem8	11110110	mod 111 r/m	F6	2−4	(107−118) +EA	(temp)←(AX) if(temp)/(mem8)>0 and(temp)/(mem8) >7FH or(temp)/(mem8)>0 and (temp)/(mem8)<0−7FH−1 then (SP)←(SP)−2, ((SP)+1 : (SP))←FLAGS,	U	U	U	U	U	U

ニーモニック	オペランド	オペレーション・コード 76543210	オペレーション・コード 76 543 210	第1オペコード バイト/ワード	バイト数	クロック数	オペレーション	フラグ O	S	Z	A	P	C
IDIV	mem8	11110110	mod 111 r/m	F6	2−4	(107−118) +EA	(IF)←0, (TF)←0, (SP)←(SP)−2, ((SP)+1:(SP))←(CS), (CS)←(2) (SP)←(SP)−2, ((SP)+1:(SP))←(IP), (IP)←(0) else (AL)←(temp)/(mem8), (AH)←(temp)%(mem8)	U	U	U	U	U	U
	reg16	11110111	11 111 r/m	F7	5	165−184	(temp)←(DX:AX) if(temp)/(reg16)>0 and(temp)/(reg16) >7FFFH or(temp)/(reg16)<0 and (temp)/(reg16)<0−7FFFH−1 then (SP)←(SP)−2, ((SP)+1:(SP))←FLAGS, (IF)←0, (TF)←0, (SP)←(SP)−2, ((SP)+1:(SP))←(CS), (CS)←(2) (SP)←(SP)−2, ((SP)+1:(SP))←(IP), (IP)←(0) else (AX)←(temp)/(reg16), (DX)←(temp)%(reg16)	U	U	U	U	U	U
	mem16	11110111	mod 111 r/m	F7	2−4	(171−190) +EA	(temp)←(DX:AX) if(temp)/(mem16)>0 and (temp)/(mem16)>7FFFH or (temp)/(mem16)<0 and(temp)/(mem16) <0−7FFFH−1 then (SP)←(SP)−2, ((SP)+1:(SP))←FLAGS, (IF)←0, (TF)←0, (SP)←(SP)−2, ((SP)+1:(SP))←(CS), (CS)←(2) (SP)←(SP)−2, ((SP)+1:(SP))←(IP), (IP)←(0) else (AX)←(temp)/(mem16), (DX)←(temp)%(mem16)	U	U	U	U	U	U
IMUL	reg8	11110110	11 101 r/m	F6	2	80−98	(AX)←(AL)*(reg8) EXT=AH, LOW=AL if(EXT)=sign extension of(LOW) then(CF)←0 else (CF)←1 ; (OF)←(CF)	×	U	U	U	U	×
	mem8	11110110	mod 101 r/m	F6	2−4	(86−104) +EA	(AX)←(AL)*(mem8) EXT=AH, LOW=AL if(EXT)=sign extension of(LOW) then(CF)←0 else (CF)←1 ; (OF)←(CF)	×	U	U	U	U	×
	reg16	11110111	11 101 r/m	F7	2 2−4	128−154	(DX:AX)←(AX)*(reg16) EXT=DX, LOW=AX if(EXT)=sign extension of(LOW) then(CF)←0 else (CF)←1 ; (OF)←(CF)	×	U	U	U	U	×
	mem16	11110111	mod 101 r/m	F7		(134−160) +EA	(DX:AX)←(AX)*(mem16) EXT=DX, LOW=AX if(EXT)=sign extension of(LOW) then(CF)←0 else (CF)←1 ; (OF)←(CF)	×	U	U	U	U	×
IN	acc, imm8	1110010W	—	E4/E5	2	10	ifW=0(AL)←(imm8) ifW=1(AX)←(imm8+1:imm8)						
	acc, DX	1110110W	—	EC/ED	1	8	ifW=0(AL)←((DX)) ifW=1(AX)←((DX)+1:(DX))						
INC	reg8	11111110	11 000 r/m	FE	2	3	(reg8)←(reg8)+1	×	×	×	×	×	
	mem	1111111W	mod 000 r/m	FE/FF	2−4	15+EA	(mem)←(mem)+1	×	×	×	×	×	
	reg16	01000reg		4X別表	1	2	(reg16)←(reg16)+1	×	×	×	×	×	
INT	3	11001100	—	CC	1	52	(SP)←(SP)−2 ((SP)+1:(SP))←FLAGS, (IF)←0, (TF)←0 (SP)←(SP)−2 ((SP)+1:(SP))←(CS), (CS)←(0EH) (SP)←(SP)−2 ((SP)+1:(SP))←(IP), (IP)←(0CH)						
	imm8	11001101	—	CD	2	51	(SP)←(SP)−2 ((SP)+1:(SP))←FLAGS, (IF)←0, (TF)←0 (SP)←(SP)−2 ((SP)+1:(SP))←(CS) (CS)←(type*4+2) (SP)←(SP)−2 ((SP)+1:(SP))←(IP), (IP)←(type*4)						
INTO	—	11001110	—	CE	1	53or4	if(OF)=1 (SP)←(SP)−2 ((SP)+1:(SP))←FLAGS, (IF)←0, (TF)←0 (SP)←(SP)−2 ((SP)+1:(SP))←(CS), (CS)←(12H) (SP)←(SP)−2 ((SP)+1:(SP))←(IP), (IP)←(10H)						

ニーモニック	オペランド	オペレーション・コード 76543210	オペレーション・コード 76 543 210	第1オペコード バイト/ワード	バイト数	クロック数	オペレーション	フラグ O	S	Z	A	P	C
IRET		11001111	----	CF	1	24	(IP)←((SP)+1:(SP)),(SP)←(SP)+2 (CS)←((SP)+1:(SP)),(SP)←(SP)+2 FLAGS←((SP)+1:(SP)),(SP)←(SP)+2	R	R	R	R	R	R
JA JNBE	short-label	01110111	----	77	2	16or4	if(CF)∨(ZF)=0 (IP)←(IP)+disp						
JAE JNB	〃	01110011	----	73	2	16or4	if(CF)=0 (IP)←(IP)+disp						
JB JNAE	〃	01110010	----	72	2	16or4	if(CF)=1 (IP)←(IP)+disp						
JBE JNA	〃	01110110	----	76	2	16or4	if(CF)∨(ZF)=1 (IP)←(IP)+disp						
JCXZ	〃	11100011	----	E3	2	18or16	if(CX)=0 (IP)←(IP)+disp						
JE JZ	〃	01110100	----	74	2	16or4	if(ZF)=1 (IP)←(IP)+disp						
JG JNLE	〃	01111111	----	7F	2	16or4	if[(SF)∀(OF)]∨(ZF)=0 (IP)←(IP)+disp [ ]は通常の演算子における( )の意味						
JGE JNL	〃	01111101	----	7D	2	16or4	if(SF)∀(OF)=0 (IP)←(IP)+disp						
JL JNGE	〃	01111100	----	7C	2	16or4	if(SF)∀(OF)=1 (IP)←(IP)+disp						
JLE JNG	〃	01111110	----	7E	2	16or4	if[(SF)∀(OF)]∨(ZF)=1 (IP)←(IP)+disp [ ]は通常の演算子における( )の意味						
(JMPS)	near-label	11101001	----	E9	3	15	(IP)←(IP)+disp						
	short-label	11101011	----	EB	2	15	(I)←(IP)+disp						
JMP	regptr16	11111111	11 100 r/m	FF	2	11	(IP)←(regptr16)						
	memptr16	11111111	mod 100 r/m	FF	2−4	18+EA	(IP)←(memptr16)						
	far-label	11101010		EA	5	15	(CS)←seg (IP)←offset						
	memptr32	11111111	mod 101 r/m	FF	2−4	24+EA	(CS)←(memptr32+2) (IP)←(memptr32)						
JNE JNZ	short-label	01110101	----	75	2	16or4	if(ZF)=0 (IP)←(IP)+disp						
JNO	〃	01110001	----	71	2	16or4	if(OF)=0 (IP)←(IP)+disp						
JNP JPO	〃	01111011	----	7B	2	16or4	if(PF)=0 (IP)←(IP)+disp						
JNS	〃	01111001	----	79	2	16or4	if(SF)=0 (IP)←(IP)+disp						
JO	〃	01110000	----	70	2	16or4	if(OF)=1 (IP)←(IP)+disp						
JP JPE	〃	01111010	----	7A	2	16or4	if(PF)=1 (IP)←(IP)+disp						
JS	〃	01111000	----	78	2	16or4	if(SF)=1 (IP)←(IP)+disp						
LAHF		10011111	----	9F	1	4	(AH)← (SF):(ZF):X:(AF):X:(PF):X:(CF)						
LDS	reg16, mem32	11000101	mod reg r/m	C5	2−4	2+EA	(reg16)←(mem32) (DS)←(mem32+2)						
LEA	reg16, mem16	10001101	mod reg r/m	8D	2−4	2+EA	(reg16)←mem16						
LES	reg16, mem32	11000100	mod reg r/m	C4	2−4	16+EA	(reg16)←(mem32) (ES)←(mem32+2)						
LOCK		11110000	----	F0	1	2	Bus Lock Prefix						
LODSB LODSW	(src-string)	1010110W	----	AC/AD	1	12or 9+13*rep	ifW=0 (AL)←((SI)) if(DF)=0 then(SI)←(SI)+1else(SI)←(SI)−1 ifW=1 (AX)←((SI)+1:(SI)) if(DF)=0 then(SI)←(SI)+2 else(SI)←(SI)−2						
LOOP	short-label	11100010	----	E2	2	17or5	(CX)←(CX)−1 if(CX)≠0 (IP)←(IP)+disp						
LOOPE LOOPZ	〃	11100001	----	E1	2	18or6	(CX)←(CX)−1 if(ZF)=and(CX)≠0 (IP)←(IP)+disp						
LOOPNE LOOPNZ	〃	11100000	----	E0	2	19or5	(CX)←(CX)−1 if(ZF)=0 and(CX)≠0 (IP)←(IP)+disp						
MOV	reg, reg	1000100W	11 reg r/m	88/89	2	2	(reg)←(reg)						
	mem, reg	1000100W	mod reg r/m	88/89	2−4	9+EA	(mem)←(reg)						
	reg, mem	1000101W	mod reg r/m	8A/8B	2−4	8+EA	(reg)←(mem)						
	mem, imm	1100011W	mod 000 r/m	C6/C7	3−6	10+EA	(mem)←(data)						
	reg, imm	1011Wreg	----	BX別表	2−3	4	(reg)←(data)						
	acc, mem	1010000W	----	A0/A1	3	10	if W=0 (AL)←(mem8)ダイレクト指定の if W=1 (AX)←(mem16)み可能						
	mem, acc	1010001W	----	A2/A3	3	10	if W=0 (mem8)←(AL) if W=1 (mem16)←(AX) 同上						
	sreg, reg16	10001110	----	8E	2	2	(sreg)←(reg16)						
	sreg, mem16	10001110	11 sreg r/m	8E	2−4	8+EA	(sreg)←(mem16)						
	reg16, sreg	10001100	mod sreg r/m	8C	2	2	(reg16)←(sreg)						
	mem16, sreg	10001100	11 sreg r/m	8C	2−4	9+EA	(mem16)←(sreg)						
MOVSB MOVSW	(dst-string, src-string)	1010010W	----	A4/A5	1	18or 9+17*rep	ifW=0 ((DI))←((SI)) if(DF)=0 then(SI)←(SI)+1, (DI)←(DI)+1 else (SI)←(SI)−1,(DI)←(DI)−1						

ニーモニック	オペランド	オペレーション・コード		第1オペコード バイト/ワード	バイト数	クロック数	オペレーション	フラグ					
		76543210	76 543 210					O	S	Z	A	P	C
MOVSB MOVSW	(dst-string, src-string)	1010010W	—	A4/A5	1	18or 9+17*rep	ifW=1 ((DI)+1:(DI))←((SI)+1:(SI)) if(DF)=0 then(SI)←(SI)+2, (DI)←(DI)+2 else (SI)←(SI)−2,(DI)←(DI)−2						
MUL	reg8	11110110	11 100 r/m	F6	2	70−77	(AX)←(AL)*(reg8) EXT=AH if(EXT)=0 then(CF)←0 else (CF)←1;(OF)←(CF)	×	U	U	U	U	×
	mem8	11110110	mod 100 r/m	F6	2−4	(76−83) +EA	(AX)←(AL)*(mem8) EXT=AH if(EXT)=0 then(CF)←0 else (CF)←1;(OF)←(CF)	×	U	U	U	U	×
	reg16	11110111	11 100 r/m	F7	2	118−133	(DX:AX)←(AX)*(reg16) EXT=DX if(EXT)=0 then(CF)←0 else (CF)←1;(OF)←(CF)	×	U	U	U	U	×
	mem16	11110111	mod 100 r/m	F7	2−4	(124−139) +EA	(DX:AX)←(AX)*(mem16) EXT=DX if(EXT)=0 then(CF)←0 else (CF)←1;(OF)←(CF)	×	U	U	U	U	×
NEG	reg	1111011W	11 011 r/m	F6/F7	2	3	(reg)←0−(reg) if destination=0 then(CF)←0 else (CF)←1	×	×	×	×	×	×
	mem	1111011W	mod 011 r/m	F6/F7	2−4	16+EA	(mem)←0−(mem) if destination=0 then(CF)←0 else (CF)←1	×	×	×	×	×	×
NOP	—	10010000	—	90	1	3	no-operation						
NOT	reg	1111011W	11 010 r/m	F6/F7	2	3	ifW=0 (reg8)←FFH−(reg8) ifW=1 (reg16)←FFFFH−(reg16)						
	mem	1111011W	mod 010 r/m	F6/F7	2−4	16+EA	ifW=0 (me8)←FFH−(mem8) ifW=1 (mem16)←FFFFH−(mem16)						
OR	reg, reg	0000100W	11 reg r/m	08/09	2	3	(reg)←(reg)∨(reg)	○	×	×	U	×	○
	mem, reg	0000100W	mod reg r/m	08/09	2−4	16+EA	(mem)←(mem)∨(reg)	○	×	×	U	×	○
	reg, mem	0000101W	mod reg r/m	0A/0B	2-4	9+EA	(reg)←(reg)∨(mem)	○	×	×	U	×	○
	reg, imm	1000000W	11 001 r/m	80/81	3-4	4	(reg)←(reg)∨ data	○	×	×	U	×	○
	mem, imm	1000000W	mod 001 r/m	80/81	3−6	17+EA	(mem)←(mem)∨ data	○	×	×	U	×	○
	acc, imm	0000110W	—	OC/OD	2−3	4	ifW=0 (AL)←(AL)∨ data 8 ifW=1 (AX)←(AX)∨ data16	○	×	×	U	×	○
OUT	imm8, acc	1110011W	—	E6/E7	2	10	ifW=0 (imm8)←(AL) ifW=1 (imm8+1:imm8)←(AX)						
	DX, acc	1110111W	—	EE/EF	1	8	ifW=0 ((DX))←(AL) ifW=1 ((DX)+1:(DX))←(AX)						
POP	mem16	10001111	mod 000 r/m	8F	2−4	17+EA	(mem16)←((SP)+1:(SP)) (SP)←(SP)+2						
	reg16	01011reg	—	5X別表	1	8	(reg16)←((SP)+1:(SP)) (SP)←(SP)+2						
	sreg	00sreg111	—	OX,IX別表	1	8	(sreg)←((SP)+1:(SP)) (SP)←(SP)+2						
POPF	—	10011101	—	9D	1	8	FLAGS←((SP)+1:(SP)) (SP)←(SP)+2	R	R	R	R	R	R
PUSH	mem16	11111111	mod 110 r/m	FF	2−4	16+EA	(SP)←(SP)−2 ((SP)+1:(SP))←(mem16)						
	reg16	01010reg	—	5X別表	1	10	(SP)←(SP)−2 ((SP)+1:(SP))←(reg16)						
	sreg	00sreg110	—	OX,IX別表	1	10	(SP)←(SP)−2 ((SP)+1:(SP))←(sreg)						
PUSHF	—	10011100	—	9C	1	10	(SP)←(SP)−2 ((SP)+1:(SP))←FLAGS						
RCL	reg, 1	1101000W	11 010 r/m	D0/D1	2	2	(tmpcf)←(CF), (CF)←high-order bit of(reg) (reg)←(reg)*2+(tmpcf) if high-order bit of(reg)←(CF) then(OF)←1 else(OF)←0	×					×
	mem, 1	1101000W	mod 010 r/m	D0/D1	2−4	15+EA	(tmpcf)←(CF), (CF)←high-order bit of(mem) (mem)←(mem)*2+(tmpct) if hihg-order bit of(mem)≠(CF) then(OF)←1 else(OF)←0	×					×
	reg, CL	1101001W	11 010 r/m	D2/D3	2	8 +4*bit	(temp)←(CL)do while(temp)≠0 (tmpcf)←(CF), (CF)←high-order bit of(reg) (reg)←(reg)*2+(tmpcf) (temp)←(temp)−1 (OF)unde fined	U					×
	mem, CL	1101001W	mod 010 r/m	D2/D3	2−4	20+EA +4*bit	(temp)←(CL) do while(temp)≠0 (tmpcf)←(CF), (CF)←high-order bit of(mem) (mem)←(mem)*2+(tmpcf) (temp)←(temp)−1 (OF)undefined	U					×
RCR	reg, 1	1101000W	11 011 r/m	D0/D1	2	2	(tmpcf)←(CF), (CF)←low-order bit of(reg),	×					×

ニーモニック	オペランド	オペレーション・コード		第1オペコード バイト/ワード	バイト数	クロック数	オペレーション	フラグ					
		76543210	76 543 210					O	S	Z	A	P	C
RCR	reg, 1	1101000W	11 011 r/m	D0/D1	2	2	(reg)←(reg)/2 high-order bit of(reg)←(tmpcf) if high-order bit of(reg)≠next-to-high-order bit of(reg) then(OF)←1 else(OF)←0	×					×
	mem, 1	1101000W	mod 011 r/m	D0/D1	2−4	15+EA	(tmpcf)←(CF), (CF)←low-order bit of(mem), (mem)←(mem)/2 high-order bit of(mem)←(tmpcf) if high-order bit of(mem)≠next-to-high-order bit of(mem)then(OF)←1 else(OF)←0	×					×
	reg, CL	1101001W	11 011 r/m	D2/D3	2	8 +4*bit	(temp)←(CL) do while (temp)≠0 (tmpcf)←(CF), (CF)←low-order bit of(reg) (reg)←(reg)/2 high-order bit of(r/m)←(tmpcf) (temp)←(temp)−1 (OF)undefined	U					×
RCR	mem, CL	1101001W	mod 011 r/m	D2/D3	2−4	20+EA +4*bit	(temp)←(CL) do while (temp)≠0 (tmpcf)←(CF), (CF)←low-order bit of(mem) (mem)←(mem)/2 high-order bit of(mem)←(tmpcf) (temp)←(temp)−1 (OF)undefined	U					×
REP REPE PEPZ	—	11110011	—	F3	1	2	do while (CX)≠0 続くバイトのプリミティブ・ストリング命令を実行する (CX)←(CX)−1 保留割込みがあれば処理する プリミティブ命令がCMPSまたはSCAS でかつ、(ZF)≠1のときループを抜出る。						
REPNE REPNZ	—	11110010	—	F2	1	2	do while (CX)≠0 続くバイトのプリミティブ・ストリング命令を実行する (CX)←(CX)−1 保留割込みがあれば処理する (ZF)≠0 のときループを抜出る						
RET	—	11000011	—	C3	1	8	(IP)←((SP)+1 : (SP)) (SP)←(SP)+2						
	optional-pop-value	11000010	—	C2	3	12	(IP)←((SP)+1 : (SP)) (SP)←(SP)+2, (SP)←(SP)+data 16						
RET (inter-segment)	—	11001011	—	CB	1	18	(IP)←((SP)+1 : (SP)) (SP)←(SP)+2 (CS)←((SP)+1 : (SP)) (SP)←(SP)+2						
	optional-pop-value	11001010	—	CA	3	17	(IP)←((SP)+1 : (SP)) (SP)←(SP)+2 (CS)←((SP)+1 : (SP)) (SP)←(SP)+2, (SP)←(SP)+data 16						
ROL	reg, 1	1101000W	11 000 r/m	D0/D1	2	2	(CF)←high-order bit of(reg), (reg)←(reg)*2+(CF) if high-order bit of(reg)≠(CF) then(OF)←1 else(OF)←0	×					×
	mem, 1	1101000W	mod 000 r/m	D0/D1	2−4	15+EA	(CF)←high-order bit of(mem), (mem)←(mem)*2+(CF) if high-order bit of(mem)≠(CF) then(OF)←1 else(OF)←0						
	reg, CL	1101001W	11 000 r/m	D2/D3	2	8 +4*bit	(temp)←(CL) do while(temp)≠0 (CF)←high-order bit of(reg), (reg)*2+(CF) (temp)←(temp)−1 (OF)undefined	U					×
	mem, CL	1101001W	mod 000 r/m	D2/D3	2−4	20+EA +4*bit	(temp)←(CL) do while(temp)≠0 (CF)←high-order bit of(mem), (mem)←(mem)*2+(CF) (temp)←(temp)−1 (OF)undefined	U					×
ROR	reg , 1	1101000W	11 001 r/m	D0/D1	2	2	(CF)←low-order bit of(reg), (reg)← (reg)←(reg)/2 high-order bit of(reg)←(CF) if high-order bit of(reg)≠next-to-high-order bit of(reg) then(OF)←1 else(OF)←0	×					×
	mem, 1	1101000W	mod 001 r/m	D0/D1	2−4	15+EA	(CF)←low-order bit of(mem), (mem)←(mem)/2 high-order bit of(mem)←(CF) if high-order bit of(mem)≠next-to-high-order bit of(mem) theh(OF)←1 else(OF)←0	×					×
	reg, CL	1101001W	11 001 r/m	D2/D3	2	8 +4*bit	(temp)←(CL) do while(temp)≠0 (CF)←low-order bit of(reg),	U					×

ニーモニック	オペランド	オペレーション・コード		第1オペコード バイト/ワード	バイト数	クロック数	オペレーション	フラグ					
		76543210	76 543 210					O	S	Z	A	P	C
ROR	reg, CL	1101001W	11 001 r/m	D2/D3	2	8 +4*bit	(reg)←(reg)/2 high-order bit of(reg)←(CF) (temp)←(temp)−1 (OF)undefined	U					×
	mem, CL	1101001W	mod 001 r/m	D2/D3	2−4	20+EA +4*bit	(temp)←(CL) do while(temp)≠0 (CF)←low-order bit of(mem), (mem)←(mem)/2 high-order bit of(mem)←(CF) (temp)←(temp)−1 (OF)undefined	U					×
SAHF	—	10011110	—	9E	1	4	(SF):(ZF):X:(AF):X:(PF):X: (CF)←(AH)		R	R	R	R	R
SAL SHL	reg, 1	1101000W	11 100 r/m	D0/D1	2	2	(CF)←high-order bit of(reg), (r/m)←(reg)*2 if high-order bit of(reg)≠(CF) then(OF)←1 else(OF)←0	×					×
	mem, 1	1101000W	mod 100 r/m	1 D0/D1	2−4	15+EA	(CF)←high-order bit of(mem), (mem)←(mem)*2 if high-order bit of (EA)≠(CF) then(OF)←1 else(OF)←0	×					×
	reg, CL	1101001W	11 100 r/m	3 D2/D3	2	8 +4*bit	(temp)←(CL), do while(temp)≠0 (CF)←high-order bit of(reg), (reg)←(reg)*2 (temp)←(temp)−1 (OF)undefined	×					×
	mem, CL	1101001W	mod 100 r/m	D2/D3	2−4	20+EA +4*bit	(temp)←(CL), do while(temp)≠0 (CF)←high-order bit of(mem), (mem)←(mem)*2 (temp)←(temp)−1 (OF)undefined	×					×
SAR	reg, 1	1101000W	11 111 r/m	D0/D1	2	2	(CF)←low-order bit of(reg), (reg)←(reg)/2, (OF)←0	×	×	×	U	×	×
	mem, 1	1101000W	mod 111 r/m	D0/D1	2−4	15+EA	(CF)←low-order bit of(mem), (mem)←(mem)/2, (OF)←0	×	×	×	U	×	×
	reg, CL	1101001W	11 111 r/m	D2/D3	2	8 +4*bit	(temp)←(CL) do while(temp)≠0 (CF)←low-order bit of(reg), (reg)←(reg)/2 (temp)←(temp)−1 (OF)undefined	× ×	× ×	× ×	U U	× ×	× ×
	mem, CL	1101001W	mod 111 r/m	D2/D3	2−4	20+EA +4*bit	(temp)←(CL) do while(temp)≠0 (CF)←low-order bit of(mem), (mem)←(mem)/2 (temp)←(temp)−1 (OF)undefined	×	×	×	U	×	×
SBB	reg, reg	0001100W	11 reg r/m	18/19	2	3	(reg)←(reg)−(reg)−(CF)	×	×	×	×	×	×
	mem, reg	0001100W	mod reg r/m	18/19	2−4	16+EA	(mem)←(mem)−(reg)−(CF)	×	×	×	×	×	×
	reg, mem	0001101W	mod reg r/m	1A/1B	2−4	9+EA	(reg)←(reg)−(mem)−(CF)	×	×	×	×	×	×
	reg, imm	1000000W	11 101 r/m	80/81	3−4	4	(reg)←(reg)−data−(CF)	×	×	×	×	×	×
		1000001W	11 101 r/m	82/83	3	4	ifW=0 (reg8)←(reg8)−data8−(CF) ifW=1 (reg16)←(reg16)−[data16←data8]−(CF)	×	×	×	×	×	×
	mem, imm	1000000W	mod 101 r/m	80/81	3−6	17+EA	(mem)←(mem)−data−(CF)	×	×	×	×	×	×
		1000001W	mod 101 r/m	82/83	3−5	17+EA	ifW=0 (mem8)←(mem8)−data8−(CF) ifW=1 (mem16)←(mem16)−[data16←data8]−(CF)	×	×	×	×	×	×
	acc, imm	0010110W	—	2C/2D	2−3	4	ifW=0 (AL)←(AL)−data 8 −(CF) ifW=1 (AX)←(AX)−data16−(CF)	×	×	×	×	×	×
SCASB SCASW	(dest-string)	1010111W	—	AE/AF	1	15or 9+15*rep	ifW=0 (AL)−((DI)) if(DF)=0 then(DI)←(DI)+1 else(DI)←(DI)−1 ifW=1 (AX)−((DI)+1:(DI)) if(DF)=0 then(DI)←(DI)+2 else(DI)←(DI)−2	×	×	×	×	×	×
SHR	reg, 1	1101000W	11 101 r/m	D0/D1	2	2	(CF)←low-order bit of(reg), (reg)←(reg)/2 if high-order bit of(reg)≠next-to-high-order bit of(reg) then(OF)←1 else(OF)←0	×					×
	mem, 1	1101000W	mod 101 r/m	D0/D1	2−4	15+EA	(CF)←low-order bit of(mem), (mem)←(mem)/2 if high-order bit of(mem)≠next-to-high-order bit of(mem) then(OF)←1 else(OF)←0	×					×
	reg, CL	1101001W	11 101 r/m	D2/D3	2	8 +4*bit	(temp)←(CL) do while(temp)≠0 (CF)←low-order bit of(reg), (reg)←(reg)/2 (temp)←(temp)−1 (OF)undefined	×					×
	mem, CL	1101001W	mod 101 r/m	D2/D3	2−4	20+EA +4*bit	(temp)←(CL) do while(temp)≠0 (CF)←low-order bit of(mem), (mem)←(mem)/2 (temp)←(temp)−1 (OF)undefined	×					×
STC	—	11111001	—	F9	1	2	(CF)←1						1
STD	—	11111101	—	FD	1	2	(DF)←1						
STI	—	11111011	—	FB	1	2	(IF)←1						

ニーモニック	オペランド	オペレーション・コード 76543210	オペレーション・コード 76 543 210	第1オペコード バイト/ワード	バイト数	クロック数	オペレーション	フラグ O	S	Z	A	P	C
STOSB STOSW	(dst-string)	1010101W	—	AA/AB	1	11or 9+10*rep	ifW=0 ((DI))←(AL) if(DF)=0 then(DI)←(DI)+1 else(DI)←(DI)−1 ifW=1 ((DI)+:(DI))←(AX) if(DF)=0 then(DI)←(DI)+2 else(DI)←(DI)−2						
SUB	reg, reg	0010100W	11 reg r/m	28/29	2	3	(reg)←(reg)−(reg)	×	×	×	×	×	×
	mem, reg	0010100W	mod reg r/m	28/29	2−4	16+EA	(mem)←(mem)−(mem)	×	×	×	×	×	×
	reg, mem	0010101W	mod reg r/m	2A/2B	2−4	9+EA	(reg)←(reg)−(mem)	×	×	×	×	×	×
	reg, imm	1000000W	11 101 r/m	80/81	3−4	4	(reg)←(reg)−data	×	×	×	×	×	×
		1000001W	11 101 r/m	82/83	3	4	ifW=0 (reg8)←(reg8)−data8 ifW=1 (reg16)←(reg16)−[data16←data8]	×	×	×	×	×	×
	mem, imm	1000000W	mod reg r/m	80/81	3−6	17+EA	(mem)←(mem)−data	×	×	×	×	×	×
		1000001W	mod reg r/m	82/83	3−5	17+EA	ifW=0 (mem8)←(mem8)−data8 ifW=1 (mem16)←(mem16)−[data16←data8]	×	×	×	×	×	×
	acc, imm	0010110W	—	2C/2D	2−3	4	ifW=0 (AL)←(AL)−data8 ifW=1 (AX)←(AX)−data16	×	×	×	×	×	×
TEST	reg, reg	1000010W	11 reg r/m	84/85	2	3	(reg)&(reg)	○	×	×	U	×	○
	mem, reg	1000010W	mod reg r/m	84/85	2−4	9+EA	(reg)&(mem)	○	×	×	U	×	○
	reg, imm	1111011W	11 000 r/m	F6/F7	3−4	5	(reg)&data	○	×	×	U	×	○
	mem, imm	1111011W	mod 000 r/m	F6/F7	3−6	11+EA	(mem)&data	○	×	×	U	×	○
	acc, imm	1010100W	—	A8/A9	2−3	4	ifW=0 (AL)& data8 ifW=1 (AX)& data16	○	×	×	U	×	○
WAIT	—	10011011	—	9B	1	3+5n	CPU wait						
XCHG	reg, reg	1000011W	11 reg r/m	86/87	2	4	(reg)↔(reg)						
	mem, reg	1000011W	mod reg r/m	86/87	2−4	17+EA	(reg)↔(mem)						
	AX, reg16	10010reg	—	9X別表	1 1	3	(reg16)↔(AX)						
XLAT	(source-table)	11010111	—	D7	1 1	11	(AL)←((BX)+(AL))						
XOR	reg, reg	0011000W	11 reg r/m	30/31	2 2	3	(reg)←(reg)∀(reg)	○	×	×	U	×	○
	mem, reg	0011000W	mod reg r/m	30/31	2−4	16+EA	(mem)←(mem)∀(reg)	○	×	×	U	×	○
	reg, mem	0011001W	mod reg r/m	32/33	2−4	9+EA	(reg)←(reg)∀(mem)	○	×	×	U	×	○
	reg, imm	1000000W	11 110 r/m	80/81	3−4	4	(reg)←(reg)∀data	○	×	×	U	×	○
	mem, imm	1000000W	mod 110 r/m	80/81	3−6	17+EA	(mem)←(mem)∀data	○	×	×	U	×	○
	acc, imm	0011010W	—	34/35	2−3	4	ifW=0 (AL)←(AL)∀ data8 ifW=1 (AX)←(AX)∀ data16	○	×	×	U	×	○

※第1オペコードで、((9X別表))などとあるものは，第1オペコード・逆アセンブル表で、上位バイトが9のものをさがして下さい、という意味です。

＊ASM86などのアセンブラでは，ジャンプ先のアドレス指定に，N88 BASICのように，ラベル名を用いることができる．したがって，"Short-label"というような指定の箇所は，ASM86などでは「ラベル名」を書き，内蔵モニタでは「アドレス」を書く．"ロケーション"というのも，ASM86などでは，メモリの位置を指定する名称を使用し，内蔵モニタでは，「アドレス」を指定する．

ハンド・アセンブルと ハンド・ディスアセンブル

ハンド・アセンブルの方法

インストラクション・セットおよび次の「アセンブル表」から，ハンド・アセンブルを行なう方法を説明します．

1　インストラクション・セットから，第1オペ・コードを見つけます．その際，即値データを持つか否かを確認します（imm指定のあるものおよび，ジャンプ命令など）．

2　インストラクション・セットにおいて，第2オペ・コードを持つものは，次ページの「アセンブル表」を用いて，第2オペ・コードをさがします．まず，左側の列から，適するアドレッシング・モードをさがし出します．

次に，上の列で，使用するレジスタを指定することにより，目的の機械語コードが見出せます(reg，regの場合以外，転送方向は考えなくてよい)．あらかじめ，regやmodが数値で与えられいるものは，その数値を用いて下さい．

3　アドレッシング・モードにディスプレースメントを持つものは，続く1または2バイトにディスプレースメントを置きます(逆ワード)．

4　即値データや，ジャンプ命令における相対アドレス指定の必要なものは次の1または2バイトにデータを置きます(逆ワード)．相対アドレスの計算は，その都度行なわなければなりません．（終了）

以上の操作を，上から順に必要な部分だけ行ないます．必要のないバイトはつめ，バイト数が，インストラクション・セットに示されたバイト数であることを確認します．

〔使用例〕 MOV　BYTE　12[BX+SI]，34 をアセンブルする．

インストラクション・セットより，MOV mem，immをさがすと，C6/C7である．バイト命令なのでC6が命令コードである．第2オペコードを持ち，mod 000 r/mの形であるから「アセンブル表」より左の列の，8[BX+SI]（ここの8は，ディスプレースメントが8ビットの意味）を見つける．上の列はregフィールドであるが，000と決まっているので，一番左側の列であり，40が第2オペコードである．続く1バイトにディスプレースメント12を置き，さらにバイトデータ34を加えて，この命令が終わる．結果は，C6　40　12　34である．

アセンブル表(第2オペランド16進変換表)

r/m	mod		reg 000 AL or AX or ES	001 CL or CX or CS	010 DL or DX or SS	011 BL or BX or DS	100 AH or SP	101 CH or BP	110 DH or SI	111 BH or DI
000	00	[BX+SI]	00	08	10	18	20	28	30	38
	01	8[BX+SI]	40	48	50	58	60	68	70	78
	10	16[BX+SI]	80	88	90	98	A0	A8	B0	B8
	11	AL or AX	C0	C8	D0	D8	E0	E8	F0	F8
001	00	[BX+DI]	01	09	11	19	21	29	31	39
	01	8[BX+DI]	41	49	51	59	61	69	71	79
	10	16[BX+DI]	81	89	91	99	A1	A9	B1	B9
	11	CL or CX	C1	C9	D1	D9	E1	E9	F1	F9
010	00	[BP+SI]	02	0A	12	1A	22	2A	32	3A
	01	8[BP+SI]	42	4A	52	5A	62	6A	72	7A
	10	16[BP+SI]	82	8A	92	9A	A2	AA	B2	BA
	11	DL or DX	C2	CA	D2	DA	E2	EA	F2	FA
011	00	[BP+DI]	03	0B	13	1B	23	2B	33	3B
	01	8[BP+DI]	43	4B	53	5B	63	6B	73	7B
	10	16[BP+DI]	83	8B	93	9B	A3	AB	B3	BB
	11	BL or BX	C3	CB	D3	DB	E3	EB	F3	FB
100	00	[SI]	04	0C	14	1C	24	2C	34	3C
	01	8[SI]	44	4C	54	5C	64	6C	74	7C
	10	16[SI]	84	8C	94	9C	A4	AC	B4	BC
	11	AH or SP	C4	CC	D4	DC	E4	EC	F4	FC
101	00	[DI]	05	0D	15	1D	25	2D	35	3D
	01	8[DI]	45	4D	55	5D	65	6D	75	7D
	10	16[DI]	85	8D	95	9D	A5	AD	B5	BD
	11	CH or BP	C5	CD	D5	DD	E5	ED	F5	FD
110	00	DIRECT	06	0E	16	1E	26	2E	36	3E
	01	8[BP]	46	4E	56	5E	66	6E	76	7E
	10	16[BP]	86	8E	96	9E	A6	AE	B6	BE
	11	DH or SI	C6	CE	D6	DE	E6	EE	F6	FE
111	00	[BX]	07	0F	17	1F	27	2F	37	3F
	01	8[BX]	47	4F	57	5F	67	6F	77	7F
	10	16[BX]	87	8F	97	9F	A7	AF	B7	BF
	11	BH or DI	C7	CF	D7	DF	E7	EF	F7	FF

レジスタ，レジスタ間の転送は，上の欄から左の欄の向き（↙）に行なわれます．

ハンド・ディスアセンブルの方法

ニーモニック一覧表(第1オペコードディスアセンブル表)

Hi \ Lo	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD b,op2 ↓	ADD w,op2 ↓	ADD b,op2 ↑	ADD w,op2 ↑	ADD AL,i	ADD AX,i	PUSH ES	POP ES	OR b,op2 ↓	OR w,op2 ↓	OR b,op2 ↑	OR w,op2 ↑	OR AL,i	OR AX,i	PUSH CS	——
1	ADC b,op2 ↓	ADC w,op2 ↓	ADC b,op2 ↑	ADC w,op2 ↑	ADC AL,i	ADC AX,i	PUSH SS	POP SS	SBB b,op2 ↓	SBB w,op2 ↓	SBB b,op2 ↑	SBB w,op2 ↑	SBB AL,i	SBB AX,i	PUSH DS	POP DS
2	AND b,op2 ↓	AND w,op2 ↓	AND b,op2 ↑	AND w,op2 ↑	AND AL,i	AND AX,i	ES :	DAA	SUB b,op2 ↓	SUB w,op2 ↓	SUB b,op2 ↑	SUB w,op2 ↑	SUB AL,i	SUB AX,i	CS :	DAS
3	XOR b,op2 ↓	XOR w,op2 ↓	XOR b,op2 ↑	XOR w,op2 ↑	XOR AL,i	XOR AX,i	SS :	AAA	CMP b,op2 ↓	CMP w,op2 ↓	CMP b,op2 ↑	CMP w,op2 ↑	CMP AL,i	CMP AX,i	DS :	AAS
4	INC AX	INC CX	INC DX	INC BX	INC SP	INC BP	INC SI	INC DI	DEC AX	DEC CX	DEC DX	DEC BX	DEC SP	DEC BP	DEC SI	DEC DI
5	PUSH AX	PUSH CX	PUSH DX	PUSH BX	POSH SP	PUSH BP	PUSH SI	PUSH DI	POP AX	POP CX	POP DX	POP BX	POP SP	POP BP	POP SI	POP DI
6	——	——	——	——	——	——	——	——	——	——	——	——	——	——	——	——
7	JO	JNO	JB JNAE	JNB JAE	JE JZ	JNE JNZ	JBE JNA	JNBE JA	JS	JNS	JP JPE	JNP JPO	JL JNGE	JNL JGE	JLE JNG	JNLE JG
8	Grp.1 b,op2,i	Grp.1 w,op2,i	Grp.1 b,op2,1 ①	Grp.1 w,op2,i ①	TEST b,op2	TEST w,op2	XCHG b,op2	XCHG w,op2	MOV b,op2 ↓	MOV w,op2 ↓	MOV b,op2 ↑	MOV w,op2 ↑	MOV sr,op2 ↑	LEA op2 ②	MOV sr,op2 ↓	POP op2 ③
9	NOP	XCHG AX,CX	XCHG AX,DX	XCHG AX,BX	XCHG AX,SP	XCHG AX,BP	XCHG AX,SI	XCHG AX,DI	CBW	CMP	CALL ℓ,4	WAIT	PUSHF	POPF	SAHF	LAHF
A	MOV AL,m⑤	MOV AX,m ⑤	MOV m,AL ⑤	MOV m,AX ⑤	MOVSB	MOVSW	CMPSB	CMPSW	TEST AL,i	TEST AX,i	STOSB	STOSW	LODSB	LODSW	SCASB	SCASW
B	MOV AL,i	MOV CL,i	MOV DL,i	MOV BL,i	MOV AH,i	MOV CH,i	MOV DH,i	MOV BH,i	MOV AX,i	MOV CX,	RET ℓ,i ⑬	MOV BX,i	MOV SP,i	MOV BP,i	MOV SI,i	MOV DI,i
C	——	——	RET i ⑬	RET	LES op2⑮	LDS op2 ⑮	MOV b,i,op2 ⑥	MOV w,i,op2 ⑥	——	——	RET ℓ,i ⑬	RET ℓ	INT 3	INT i ⑦	INTO	IRET
D	Grp.2 ⑧ b,op2,1	Grp.2 ⑧ w,op2,1	Grp.2 b,op2,CL ⑧	Grp.2 w,op2,CL ⑧	AAM⑨	AAP ⑨	——	XLAT	ESC0 op2,⑩	ESC1 op2 ⑩	ESC2 op2 ⑩	ESC3 op2 ⑩	ESC4 op2 ⑩	ESC5 op2 ⑩	ESC6 op2 ⑩	ESC7 op2 ⑩
E	LOOPNZ LOOPNE	LOOPZ LOOPE	LOOP	JCXZ	IN AL ⑪	IN AX ⑪	OUT AL ⑪	OUT AX ⑪	CALL ⑫	JMP ⑫	JMP ℓ ④	JMPS	IN AL,DX	IN AX,DX	OUT DX,AL	OUT DX,AX
F	LOCK	——	REPNE REPNZ	REP REPZ REPE	HLT	CMC	Grp.3 b,op2	Grp.3 w,op2	CLC	STC	CLI	STI	CLD	STD	Grp.4 b,op2 ⑭	Grp.4 ⑭ b,op2

1バイト命令.
2, (3)バイト目にデータを持つ.
第2オペランドを持つ.
次の1バイトに相対化されたアドレスを持つ(注のあるものを除く).
第2オペランドに命令を含む.

第2オペコードのregフィールドによる命令の一覧

reg=	000	001	010	011	100	101	110	111
Grp.1	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
Grp.2	ROL	ROR	RCL	RCR	SHL	SHR	——	SAR
Grp.3	TEST	——	NOT	NEG	MUL	IMUL	DIV	IDIV
Grp.4	INC	DEC	CALL ⑭	CALL ℓ ⑭	JMP ⑭	JMP ℓ ⑭	PUSH	——

〔記号の意味〕 b：バイト命令. w：ワード命令. op2：第2オペコードを持つ. i：即値データ(8/16ビット).
m：メモリ・アドレス. ℓ：セグメント間分岐. sr：セグメントレジスタを使用した転送.
↓：第2オペコードで指定されるデータの転送方向が上段から下段に向かっている.
↑：第2オペコードで指定されるデータの転送方向が下段から上段に向かっている.

〔注〕 ①82, 83では, 即値データが符号縮小されている. すなわち, この命令での即値データは1バイトであり, 83で始まる命令では, この1バイトデータが符号拡張された2バイトデータとして演算, 転送される. 82で始まる命令は80で始まる命令とまったく同じである.
②op2表の上段がEAのロードされるレジスタ, 下段がEAの算出されるアドレッシングである.
③op2表の下段のみ使用.
④続く2バイトが新しいIPの値, さらに続く2バイトが新しいCSの値である.
⑤続く2バイトがダイレクト・メモリ・アドレスの値である. すなわち, MOV AX, [123 4] 等の場合のみに使用される.
⑥第2オペランドで指定したアドレス(下段のみ有効)に, 即値データを転送する.
⑦続く1バイトがタイプNo.
⑧シフト数が1かCLレジスタが表中に示す.
⑨この命令は, 次の1バイトが0Aと定まった2バイト命令である.
⑩op2表の下段のみ有効, ESCシーケンスのコマンドは別の資料を参照のこと.
⑪続く1バイトがポート番号.
⑫続く2バイトが相対化された分岐アドレス.
⑬次く2バイトがSPの値に加えられる.
⑭op2表下段のメモリ/レジスタによる間接アドレス指定.
⑮op2表下段の内容が, 上段のレジスタおよびセグメントレジスタに転送される.

オペランド一覧表(第２オペコードディスアセンブル表)

Hi \ Lo	0	1	2	3	4	5	6	7
0	AL.AX.ES(ADD*.ROL.TEST*.INC)							
	[BX+SI]	[BX+DI]	[BP+SI]	[BP+DI]	[SI]	[DI]	DIRECT	[BX]
1	DL.DX.SS(ADC*.---.NOT.CALL id)							
	[BX+SI]	[BX+DI]	[BP+SI]	[BP+DI]	[SI]	[DI]	DIRECT	[BX]
2	AH.SP (AND*.SHL.MUL.JMP id)							
	[BX+SI]	[BX+DI]	[BP+SI]	[BP+DI]	[SI]	[DI]	DIRECT	[BX]
3	DH.SI (XOR*.---.DIV.PUSH)							
	[BX+SI]	[BX+DI]	[BP+SI]	[BP+DI]	[SI]	[DI]	DIRECT	[BX]
4	AL.AX.ES(ADD*.ROL.TEST*.INC)							
	8[BX+SI]	8[BX+DI]	8[BP+SI]	8[BP+DI]	8[SI]	8[DI]	8[BP]	8[BX]
5	DL.DX.SS(ADC*.---.NOT.CALL id)							
	8[BX+SI]	8[BX+DI]	8[BP+SI]	8[BP+DI]	8[SI]	8[DI]	8[BP]	8[BX]
6	AH.SP (AND*.SHL.MUL.JMP id)							
	8[BX+SI]	8[BX+DI]	8[BP+SI]	8[BP+DI]	8[SI]	8[DI]	8[BP]	8[BX]
7	DH.SI (XOR*.---.DIV.PUSH)							
	8[BX+SI]	8[BX+DI]	8[BP+SI]	8[BP+DI]	8[SI]	8[DI]	8[BP]	8[BX]
8	AL.AX.ES(ADD*.ROL.TEST*.INC)							
	16[BX+SI]	16[BX+DI]	16[BP+SI]	16[BP+DI]	16[SI]	16[DI]	16[BP]	16[BX]
9	DL.DX.SS(ADC*.---.NOT.CALL id)							
	16[BX+SI]	16[BX+DI]	16[BP+SI]	16[BP+DI]	16[SI]	16[DI]	16[BP]	16[BX]
A	AH.SP (AND*.SHL.MUL.JMP id)							
	16[BX+SI]	16[BX+DI]	16[BP+SI]	16[BP+DI]	16[SI]	16[DI]	16[BP]	16[BX]
B	DH.SI (XOR*.---.DIV.PUSH)							
	16[BX+SI]	16[BX+DI]	16[BP+SI]	16[BP+DI]	16[SI]	16[DI]	16[BP]	16[BX]
C	AL.AX.ES(ADD*.ROL.TEST*.INC)							
	AL.AX	CL.CX	DL.DX	BL.BX	AH.SP	CH.BP	DH.SI	BH.DI
D	DL.DX.SS(ADC*.---.NOT.CALL id)							
	AL.AX	CL.CX	DL.DX	BL.BX	AH.SP	CH.BP	DH.SI	BH.DI
E	AH.SP (AND*.SHL.MUL.JMP id)							
	AL.AX	CL.CX	DL.DX	BL.BX	AH.SP	CH.BP	DH.SI	BH.DI
F	DH.SI (XOR*.---.DIV.PUSH)							
	AL.AX	CL.CX	DL.DX	BL.BX	AH.SP	CH.BP	DH.SI	BH.DI

Hi \ Lo	8	9	A	B	C	D	E	F
0	CL.CX.CS(OR* .ROR.---.DEC)							
	[BX+SI]	[BX+DI]	[BP+SI]	[BP+DI]	[SI]	[DI]	DIRECT	[BX]
1	BL.BX.DS(SBB*.RCR.NEG.CALL l-id)							
	[BX+SI]	[BX+DI]	[BP+SI]	[BP+DI]	[SI]	[DI]	DIRECT	[BX]
2	CH.BP (SUB*.SHR.IMUL.JMP l-id)							
	[BX+SI]	[BX+DI]	[BP+SI]	[BP+DI]	[SI]	[DI]	DIRECT	[BX]
3	BH.DI (CMP*.SAR.IDIV.---)							
	[BX+SI]	[BX+DI]	[BP+SI]	[BP+DI]	[SI]	[DI]	DIRECT	[BX]
4	CL.CX.CS(OR* .ROR.---.DEC)							
	8[BX+SI]	8[BX+DI]	8[BP+SI]	8[BP+DI]	8[SI]	8[DI]	8[BP]	8[BX]
5	BL.BX.DS(SBB*.RCR.NEG.CALL l-id)							
	8[BX+SI]	8[BX+DI]	8[BP+SI]	8[BP+DI]	8[SI]	8[DI]	8[BP]	8[BX]
6	CH.BP (SUB*.SHR.IMUL.JMP l-id)							
	8[BX+SI]	8[BX+DI]	8[BP+SI]	8[BP+DI]	8[SI]	8[DI]	8[BP]	8[BX]
7	BH.DI (CMP*.SAR.IDIV.---)							
	8[BX+SI]	8[BX+DI]	8[BP+SI]	8[BP+DI]	8[SI]	8[DI]	8[BP]	8[BX]
8	CL.CX.CS(OR* .ROR.---.DEC)							
	16[BX+SI]	16[BX+DI]	16[BP+SI]	16[BP+DI]	16[SI]	16[DI]	16[BP]	16[BX]
9	BL.BX.DS(SBB*.RCR.NEG.CALL l-id)							
	16[BX+SI]	16[BX+DI]	16[BP+SI]	16[BP+DI]	16[SI]	16[DI]	16[BP]	16[BX]
A	CH.BP (SUB*.SHR.IMUL.JMP l-id)							
	16[BX+SI]	16[BX+DI]	16[BP+SI]	16[BP+DI]	16[SI]	16[DI]	16[BP]	16[BX]
B	BH.DI (CMP*.SAR.IDIV.---)							
	16[BX+SI]	16[BX+DI]	16[BP+SI]	16[BP+DI]	16[SI]	16[DI]	16[BP]	16[BX]
C	CL.CX.CS(OR* .ROR.---.DEC)							
	AL.AX	CL.CX	DL.DX	BL.BX	AH.SP	CH.BP	DH.SI	BH.DI
D	BL.BX.DS(SBB*.RCR.NEG.CALL l-id)							
	AL.AX	CL.CX	DL.DX	BL.BX	AH.SP	CH.BP	DH.SI	BH.DI
E	CH.BP (SUB*.SHR.IMUL.JMP l-id)							
	AL.AX	CL.CX	DL.DX	BL.BX	AH.SP	CH.BP	DH.SI	BH.DI
F	BH.DI (CMP*.SAR.IDIV.---)							
	AL.AX	CL.CX	DL.DX	BL.BX	AH.SP	CH.BP	DH.SI	BH.DI

「ニーモニック一覧表」と，「オペランド一覧表」を用いて，ハンド・ディスアセンブルを行なう方法を説明します．

機械語コードを命令とみなし，「ニーモニック一覧表」から，命令を見つけます．その際，命令が次のいずれであるかに基づいて，続くコードのディスアセンブルを行ないます．

1．1バイト命令．
2．データまたは相対アドレスを持つ命令．
3．第2オペコードを持つ命令．
4．第2オペコードに命令を含む命令（第nグループ）．

●データまたは相対アドレスを持つ命令の場合●

「ニーモニック一覧表」に示されたバイト数（bまたはwなど）が後続しているので，それをデータまたはアドレスに変換する．（終了）

●第2オペコードを持つ命令の場合●

次のバイトから，「オペランド一覧表」を用いて，使用するメモリまたはレジスタを求めます．上段が使用されるレジスタ（横8欄共通．（　）内は関係ない）下段が使用されるメモリまたはレジスタであり，扱うバイト数（ニーモニック一覧表のb, w）によって，レジスタのビット数を決めます．転送方向は，「ニーモニック一覧表」の↑↓の方向です．

使用するメモリのアドレッシング・モードにディスプレースメントを含むものは続く1または2バイトがディスプレースメントになります．（終了）

●第2オペコードに命令を含む命令(第nグループ)の場合●

次のバイトから，「オペランド一覧表」を用いて該当する場所をさがします．上段の（　）内の命令のうち，左からn番めのものが該当する命令であり，下段のメモリまたはレジスタが命令の作用するメモリまたはレジスタです．

使用するメモリのアドレッシング・モードにディスプレースメントを含むものは，続く1または2バイトがディスプレースメントになります．

命令末尾に＊印の付いた命令では，さらに1バイトまたは2バイトのデータが続きます（命令コードが83のものは，1バイトデータが符号拡張されるので注意）．

〔使用例〕 81　47　10　34　12　50……をディスアセンブルする．

「ニーモニック一覧表」より81は第1グループに属したワード命令であることがわかる．そこで次のバイトを「オペランド一覧表」を用いて調べると，47の位置は上段が　AX, AL, ES（ADD＊，ROL，TEST＊，INC）　であり，下段が8[BX]である．第1グループに属しているので，（　）の中で1番目のADD命令である．また，使用するメモリはBXレジスタによる間接指定で，8ビットのディスプレースメントを持っていることがわかるから，次の1バイトの10はディスプレースメントである．さて，本当ならこれで終わりだが，ADD命令の末尾に＊がついているので，さらにデータ（2バイト）を持つ．したがって次の34，12がデータである．以上からこの命令は，ADD　　WORD　10[BX]，1234　となる．

さ く い ん

〈数　字〉

0

0でないデータを見つける……202
0による除算……124

1

10進補正命令……132, 254
16進数……42, 46
16ビット……18
16ビットCPU……23
16ビットを越える加減算……100, 109

2

2進化10進数……253
2進数……42
2進数の加減算……44
2の補数……48

3

3(トライ)ステート……34

6

68000……24
68系……24

8

8085メモリモデル……85
8086……14, 23
8086の端子接続……35
80系……23
8ビットCPU……23
8ビットから16ビットへの変換……13

〈英　語〉

A

AAA……255
AAD……255
AAM……255
AAS……255
ADC……100, 101
ADD……96, 97
AF(補助キャリー・フラグ)……132
ALU……31
AND……238, 239, 242
AND回路……33
ASM86……51, 55
Aコマンド……62

B

BASICにもどる……130
BCDコード……253
BIOS……251
BLOAD……74
BSAVE……72

C

CALL……178, 180
CBW……116, 117
CF(キャリー・フラグ)……131
CLC……134
CLD……134
CLEAR文……21
CLI……134, 235
CMC……134
CMP……142, 158
CMPS……218
CMPSB……224
CP/M-86……51, 55

CPU……17, 18, 23
CPU制御命令……259
CPUの内部構成……30
CWD……116, 117
Cコマンド……58

D

DAA……255
DAS……255
DEC……138, 158
DEC社……24
DEF SEG文……73
DF(ディレクション・フラグ)……133
DIV……122, 126
Dコマンド……58

E

ESC……259, 260
Eコマンド……68

F

FIFO……32
Fコマンド……69

G

Gコマンド……64

H

HLT……259, 260

I

I/O……16
I/Oポート……250
I/Oマップド I/O……252
I/O命令……250
IDIV……122, 126
IF(インタラプト・イネーブル・フラグ)……133
IMUL……120, 120, 121
IN……252
INC……141, 158
INT……168, 231, 235
INT 0……124
INT 1……133
INT 1Eの使い方……130
INTO……235
INTR端子……133
IRET……180, 235

J

JA……157
JAE……157
JB……157
JBE……157
JCXZ……157
JE……157
JG……157
JGE……157
JL……157
JLE……157
JMP……148, 156
JMPS……156
JNA……157
JNAE……157
JNB……157
JNBE……157
JNE……157
JNG……157
JNGE……157
JNL……157
JNLE……157
JNO……157
JNP……157
JNS……157
JNZ……157
JO……157
JP……157
JPE……157
JPO……157
JS……157
JZ……157

L

LAHE……134
LDS……257, 258
LEA……256, 258
LES……257, 258
LOCK……259, 260
LODS……208

LODSB/W……224
LOOP……207
LOOPE……201, 207
LOOPNE……201, 207
LOOPNZ……201, 207
LOOPZ……201, 207
LSB……45, 111
Lコマンド……59

M

MOD……167
MOV……76, 79
MOVS……212
MOVSB/W……224
MPU……23
MS-DOS……51
MSB……45, 111
MUL……120, 121
Mコマンド……69

N

NEG……241, 242
NMI……39
NMI端子……133
NOP……260
NOPの使い方……130
NOT……238, 239, 242
O

O

OF(オーバーフロー・フラグ)……132
OR……238, 240, 242
OUT……252

P

PC-9801……14
PC-9801のブロック図……16
PEEK……132
PF(パリティ・フラグ)……132
POKE……166
POP……187, 188
POPF……134, 188
PUSH……187, 188
PUSHF……134, 188
Pコマンド……71

R

RAM……17
RAMの使用法……21
RCL……245, 248
RCR……245, 248
REP……218
REPE……218
REPNE……218
REPNZ……218
REPZ……218
RET……178, 180
ROL……245, 248
ROM……17
ROM内ルーチンコール……231
ROR……245, 248

S

SAHF……134
SAL……245, 247
SAR……245, 248
SBB……108, 109
SCAS……218
SCASB/W……224
SF(サイン・フラグ)……132
SHL……245, 247, 248
SHR……245, 247
STC……134
STD……134
STI……134, 235
STOS……208
STOSB/W……224
SUB……107
Sコマンド……65

T

TEST……241, 242
TF(トラップ・フラグ)……133
TTL……33

U

USR関数……168
USR関数の使い方……169

V

VRAM……18

W

WAIT……259, 260

X

XCHG……103
XLAT……257, 258
XOR……238, 240, 242
Xコマンド……69

Z

ZF(ゼロ・フラグ)……132

〈日本語〉

ア

アキュムレータ・レジスタ……26
アスキー・コード……232
アセンブラ……55, 62
アセンブラ中の計算式……112
アセンブリ言語……51
アセンブル……51
アドレッシング・モード……87, 90
アドレス……19
アドレス情報……35
余りの符号……123
アンパックド10進数……255
暗黙のオペランド指定……208
暗黙のセグメント指定……90
イネーブル……37, 230
イミディエイト……89
イミディエイト・モード……87
インストラクション・ポインタ・レジスタ……26, 27, 78
インター・フェイス……17
インター・プリタ……50
インタラプト・イネーブル・フラグ……133
インテル社……23
インデックス・レジスタ……88
ウェイト・ルーチン……252
エクストラ・セグメント……27
演算機能……18
オーバーフロー……115, 125
オーバーフロー・フラグ……132, 183, 235
オーバーフロー割り込み……229
オフセット・アドレス……28
オペランド……51, 76, 96

カ

カウンタ・レジスタ……26, 141
加算命令……96
関数……166
間接指定法……77
間接モード……87
画面出力の方法……233
キャリー……100
キャリーを含むローテート命令……245
キャリー・フラグ……98, 131
キャリー・フラグの使用例……206
キュー……31
キューの状態……39
キーボード入力……231
機械語……50
機械語エリア……21
機械語ルーチン……168
逆ワード……82
クリア……99, 240
繰り返しの必要なプログラム……136, 198
繰り返し回数の設定……194
クロック……40
減算命令……107
コード・セグメント……27, 78
高級言語……176
異なるデータを見つける方法……219
コマンド……166
コントロール・フラグ……133
コンパイラ……50

サ

サイン・フラグ……132
先のアドレスへの分岐法……145
サブルーチン……175
サブルーチンがスタックを使用する理由……191
サブルーチンとスタック……183
算術・論理演算……31
ザイログ社……23
商……167

商の符号……123
システム・エリア……20
システム・ディスク……56
システム・ポート……17
シフト命令……244
シングル・ステップ割り込み……230
真理値表……33
条件判断を伴うループ命令……200
条件分岐命令……137, 141, 153, 156
乗算命令……120, 121
乗除算命令……246
剰余……167
除算命令……122, 181
実行ユニット……30
実効アドレス……256
時分割……35
数値データ・プロセッサ……17
スタック……183
スタック・セグメント……27
スタック・ポインタ……26, 184
スタックの用途……192
スタックへの読み書き……187
ステータス情報……35
ステータス・フラグ……131
ストリング・命令……208, 209, 213
セット……99
正論理……33
制御機能……18
制御命令……259
セグメンテーション……29
セグメント・オーバーライド・プリフィックス…78, 85
セグメント・レジスタ……26
セグメント間分岐命令……152
セグメント指定……90
セグメントの概念……27
セマフォ……259
絶対アドレス……151
絶対分岐法……155
ゼロ・フラグ……132
ゼロディバイド割り込み……229
前置命令……213
ソース……76
ソース・インデックス……27, 105
相対アドレス指定……151
相対アドレスの計算法……163
相対分岐法……154
即値……89
素数を求めるプログラム……172
ソフトウエア割り込み……133, 228

タ

タイマ……226
ダイレクト……89
直接アドレス指定法……150, 178
直接モード……87
チップ・セレクト信号……37
転送命令……76
転送命令群……214
ディスアセンブル……51, 59
ディスティネーション……27, 76, 105
ディスプレースメント……88, 164
ディレクション・フラグ……209
データ情報……35
データ・セグメント……27, 78
データ・レジスタ……26
データの退避……69
データ転送命令……76
データのカウント……200
デシマル……43
特殊な内部割り込み……229
トグル・プリンタ・スイッチ……71
トラップ・フラグ……133
トラップ・フラグの変更法……242

ナ

内部ブロック図……31
内部割り込み……133, 228
入出力装置……15, 16, 250
入出力と割り込み……231
入力の修正法……60, 64
ニーモニック……51, 76, 96
ノン・マスカブル・インタラプト……230

ハ

ハードウエア……14
ハイ・インピーダンス……34
排他的論理和……238
反転……243
ハンド・アセンブル……54
汎用レジスタ……26
バイト……19
バイト指定……83
バイナリー……43

バス……15
バス・インターフェイス・ユニット……30
バス・コントローラ……38
バス・ハイ・イネーブル……36
番地……19
パックド10進数……255
パリティ・フラグ……132
比較……144
比較命令……136, 158
比較命令群……214
否定……238
ビット……19, 45
ビット状態の調査……241
負論理……33
フェッチ……32
復帰……179
副プログラム……175
符号拡張命令……115, 181
符号付演算……111, 115, 132
符号付き分岐命令……145
符号無し分岐命令……145
符号反転……241
符号ビット……111, 115
負数の入力法……112, 121
負の数の表現……47, 111
フラグ・レジスタ……27, 71, 131, 141
ブザーを鳴らす……251
物理アドレス……28
ブレークポイント……130, 229
ブロック化……175
ブロック転送……69
分岐先のアドレス指定法……150
分岐命令……136, 156
分岐命令の種類……144
プリ・フェッチ……32
プリ・フィックス……213
プログラムの退避法……69
プロンプト……71
平均値の求め方……180
ヘルプキー……72
ベース・ポインタ……26
ベース・レジスタ……26, 88, 105
補助キャリー・フラグ……132
補数……47, 241
補数表現と負の数……111
ボロー……100, 108, 131
ポート……250

マ

マイコンの基本構成……15
マキシマム・モード……37
マシン語……50
マシン語のロード・セーブ……72
マスク……239
ミニ・コンピュータ……24
ミニマム・モード……37
無条件分岐命令……137, 148, 156
命令キュー……31
命令長……53
メモリ……25
メモリ・アドレッシング……88, 90
メモリ・マップ……20, 22
メモリ・マップド・I/O……252
メモリとアドレス……19
メモリのクリア法……141
メモリポインタ間接指定法……151, 178
文字出力……233
モニタ……55
モニタ・コマンド……57
リセット……39, 99

ラ

リセット……39, 99
リピート・プリフィックス命令……213, 218
リロケータブル……110
ループ命令……198
レジスタ……18, 25
レジスタ・アドレッシング……88
レジスタの退避……189
レジスタの表示……70
レジスタの復帰……189
レジスタ・ポインタ・間接指定法……151, 178
ロー・アクティブ……37
ローテート命令……244
論理アドレス……28
論理回路……33
論理演算命令……238
論理積……238
論理和……238

ワ

ワークエリア……20
ワード……19
ワード指定……83

割り込みイネーブル……230, 235
割り込みコントローラ……17, 230
割り込み処理ルーチン……179, 225
割り込みタイプ……227
割り込みの禁止……230
割り込みの使用法……226
割り込みベクトル……20, 227
割り込みポインタ・テーブル……227, 235
割り込み要求……225

参考文献

「iAPX 86, 88User's Manual」intel(英文)：CQ出版社
「iAPX 86/88, 186-188User's Manual Programmer's Reference」intel(英文)：CQ出版社取次
「iAPX 86マクロアセンブリ言語プログラミングマニュアル」インテルジャパン：CQ出版社取次
「マイクロコンピュータシリーズ15 8086マイクロコンピュータ」田辺皓正著：丸善

98 FAN BOOKS 2

98 マシン語

著者　藤木文彦

昭和59年8月25日　初版第1刷発行

著　者　藤木文彦
発行者　片岡　巌
発行所　技術評論社
東京都千代田区平河町1-4-12
電話　03(262) 9351(営業部直通)
03(237) 8315(編集部直通)
印刷・製本　図書印刷

ISBN4-87408-262-9 C3055 ¥2800E
定価はカバーに表示してあります。

本書の一部または全部を著作権法の定める範囲を超え，無断で複写，複製，転載，テープ化，ファイルに落すことを禁じます。

藤木文彦©

98
FAN
BOOKS

ISBN4-87408-262-9 C3055 ¥2800E

定価 2800円